www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introduction to OmpSs-2

Rosa M. Badia, Xavier Teruel, Xavier Martorell

Barcelona, September 6th, 2022

Introduction to StarSs

The StarSs "Granularities"

StarSs

OmpSs @ SMP, @ GPU, @ FPGA, @ Cluster	COMPSs PyCOMPSs
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++
Courses: October '22 March '23	Courses: January '23

Parallel ⟷ **Ensemble, workflow**

The parallel programming revolution

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

Schedule @ programmers mind

Static

Complexity: Increasing divergence between our mental model and reality

System variability

Schedule @ system

Dynamic

- 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

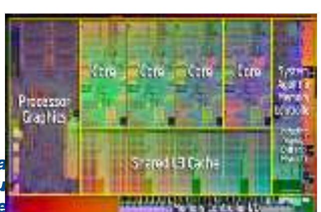
- Need to decouple again

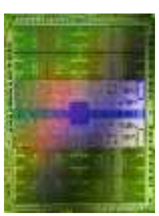
Applications

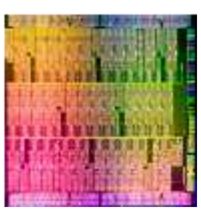
PM: High-level, clean, abstract interface

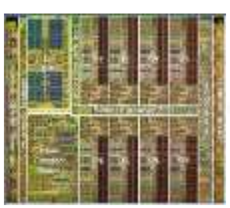
Power to the runtime

ISA / API









Application logic

Arch. independent

General purpose
Task based
Single address space

“Reuse”
architectural ideas
under
new constraints

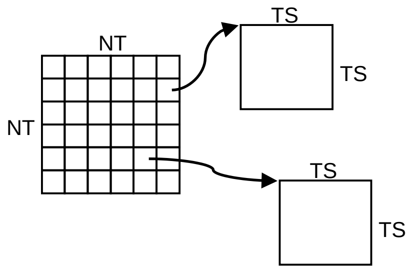
Address spaces
(hierarchy, transfers)
Control flow (cores,
threads, accelerators)

OmpSs Concept

The StarSs family of programming models

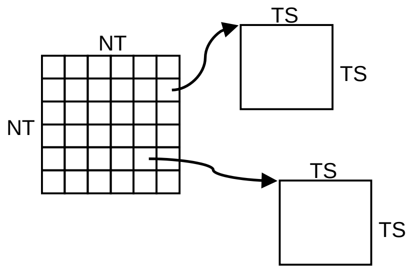
- Key concept
  - **Sequential task based** program on **single address/name space + directionality annotations**
  - Happens to execute parallel: Automatic run time computation of dependencies between tasks
    - ~ Sequential equivalence
- StarSs main characteristics
  - Dependences: Tasks instantiated 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

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 …

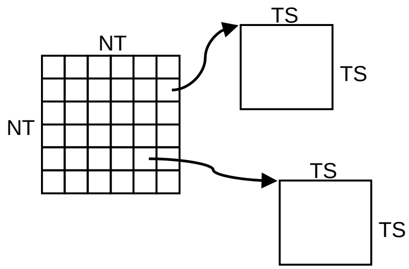


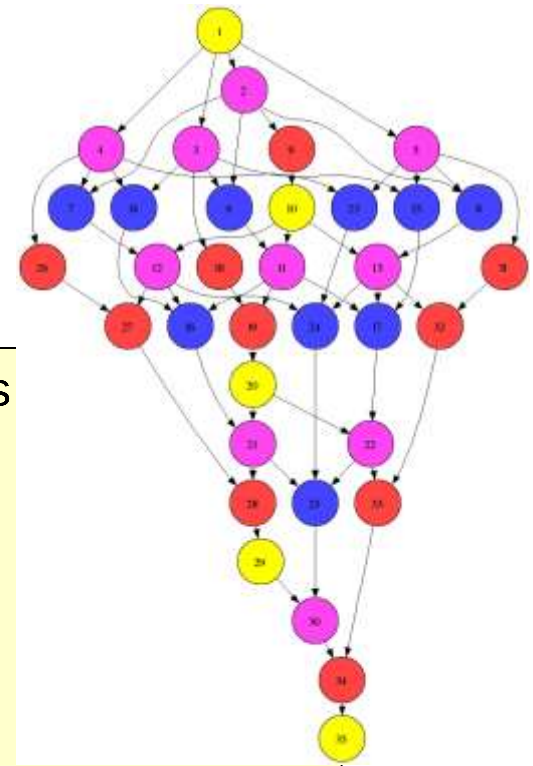
OmpSs-2 annotations

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

StarSs: … that happens to execute in parallel



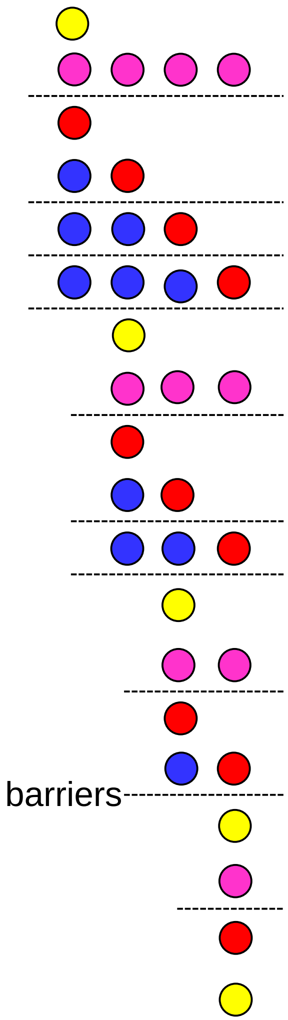


OmpSs-2 annotations

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

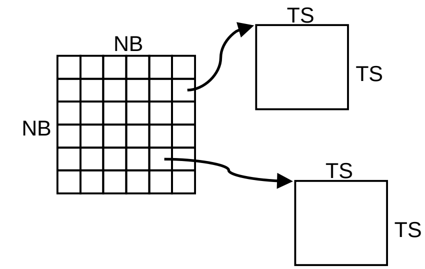
Decouple how we write/think (sequential) from how it is executed

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
   }
}
```

OpenMP annotations: tasks



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]);
 }
}
```

OpenMP annotations worksharings

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Compiling OmpSs-2 programs

- Compiling

```
mcc --ompss-2  -c bin.c
```

- Linking

```
mcc --ompss-2 -o bin bin.o
```

- Compiler accepts several GCC options

- Compatibility with OpenMP
  - --openmp-compatibility
  - Accepts OmpSs/OpenMP directives
    - #pragma omp …
  - Do not use in this course. Let the omp directives for the GCC compiler
    - e.g., OMP SIMD

Related Models and environments

- Programming models
  - OpenMP 4.5
  - Cilk++
  - TBB
  - OpenACC
  - CUDA
  - OpenCL
  - PGAS models
    - UPC
    - X10
    - Chapel

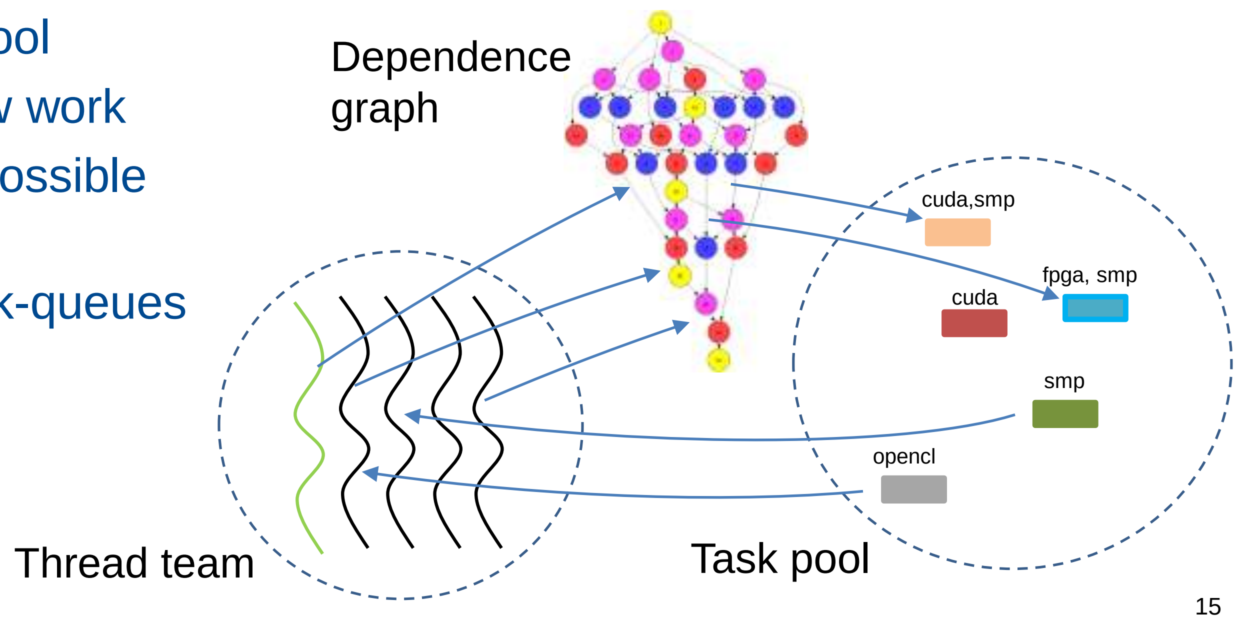
- Runtimes
  - ParalleX, HPX @ LSU
  - Swarm
  - StarPU @ U. Bordeaux
  - PLASMA, … @ UTK
  - Argobots @ ANL
  - Open Community Runtime
  - …

Basic OmpSs-2

Execution Model

- Thread-pool model
  - OpenMP parallel not supported (it was ignored in OmpSs-1)
- All threads created on startup
  - One of them starts executing main (on SMP device)
  - P-1 workers execute SMP tasks
    - $ taskset –c 0-$((P-1))
    - Equivalent to the OpenMP OMP_NUM_THREADS
  - One representative (OpenCL/CUDA/FPGA) per device/accelerator

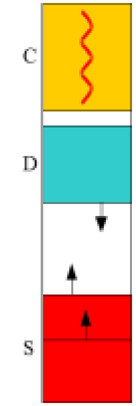
- All get work from a task pool
  - And can generate new work
  - Work is labeled with possible "targets"
  - Single or multiple work-queues



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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 oss 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

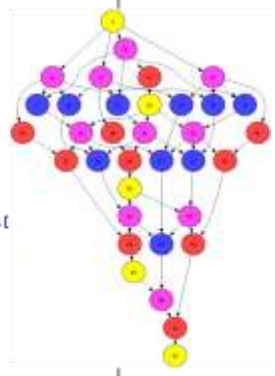
- Relaxations:
  - No need to specify for ALL data read/written, can use representatives
  - Order in inout chains
    - Programmer responsible of possible races

```
#pragma oss concurrent (…)
#pragma oss critical | atomic

#pragma oss commutative (…)
```

Inlined and outlined tasks

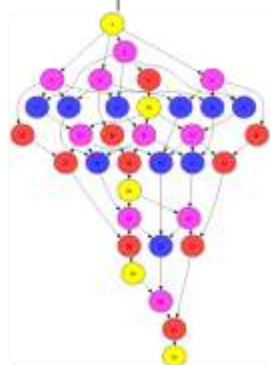
- Pragmas inlined
  - Pragma applies to immediately following statement
  - The compiler outlines the statement (as in OpenMP)

```
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);
      }
   }
}
```



- Pragmas 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);
      }
   }
}
```



BSC
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Inlined directives

Inlined task instantiation and wait

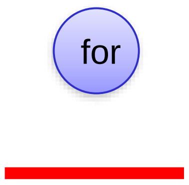
`#pragma oss task`

- Instantiates a task and control flow continues

`#pragma oss taskwait`

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

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

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

   ...
}
```



Inlined task instantiation and wait

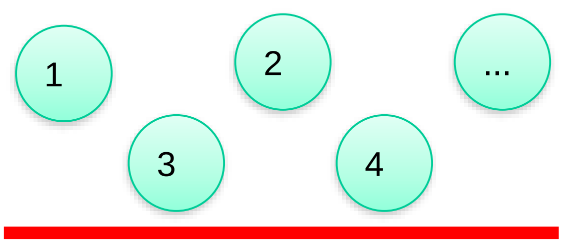
#pragma oss task

- Instantiates a task and control flow continues

#pragma oss 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 oss task
      process ( e ) ;
   }

#pragma oss 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

Dependences

```
#pragma oss 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

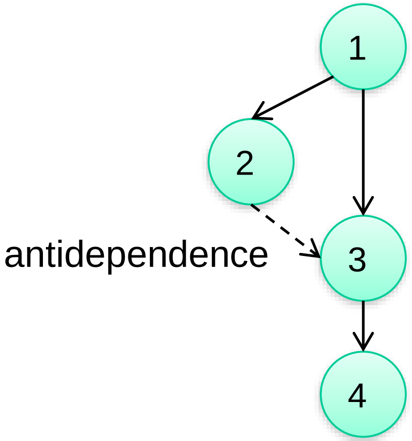
- Relaxations:
  - No need to specify for ALL data read/written, can use representatives
  - Order of inout chains
    - Programmer responsible of possible races

```
#pragma oss concurrent (…)
   #pragma oss critical | atomic

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



OpenMP: Dependencies

- Dependences are now supported in OpenMP 4.0/4.5/5.0/5.1
  - Different syntax but same semantics

```
#pragma oss task [ depend (in: var_list) ] [ depend(out: var_list) ] [ depend(inout: var_list) ]
{
    // code block
}

    //        var_list:   var_name    |   var_name[start:size]…

    //        OpenMP  inoutset (concurrent) and mutexinoutset (commutative) not supported yet
```

"OpenMP Application Program Interface. Version 4.0 " July 2013

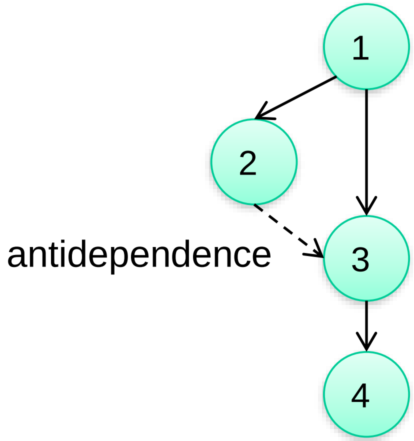
and

"OpenMP Application Program Interface. Version 4.5" November 2015

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Defining dependences for inlined tasks in OpenMP 4.0

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

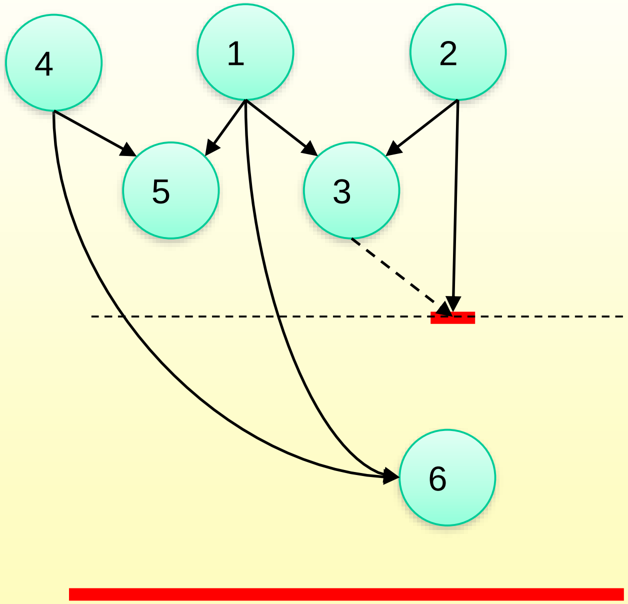


Partial control flow synchronization (OmpSs)

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

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

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

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

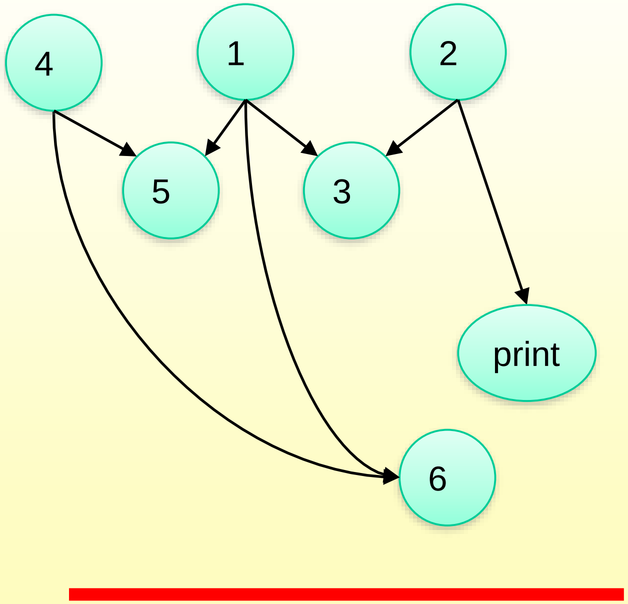


Partial control flow synchronization (OmpSs & OpenMP)

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

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

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

  #pragma oss 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

  OmpSs

  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)

  OmpSs

  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)

  OmpSs

Array sections

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

=

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

=

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

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

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

=

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

Array sections

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

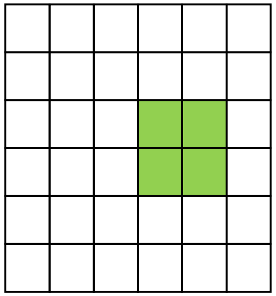
=

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

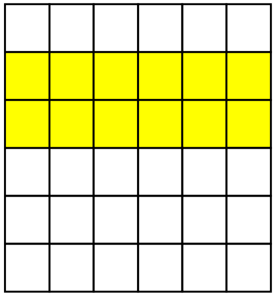
```
int a[N][M];
#pragma oss task in(a[2:3][3:4])
// 2 x 2 subblock of a at a[2][3]
```

=

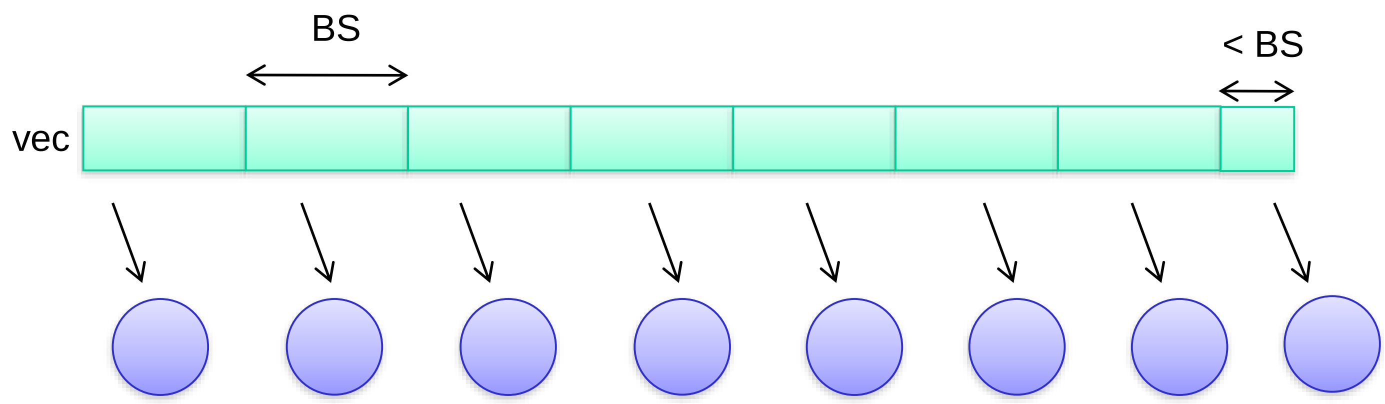
```
int a[N][M];
#pragma oss task in(a[2;2][3;2])
// 2 x 2 subblock of a at a[2][3]
```



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



Array sections

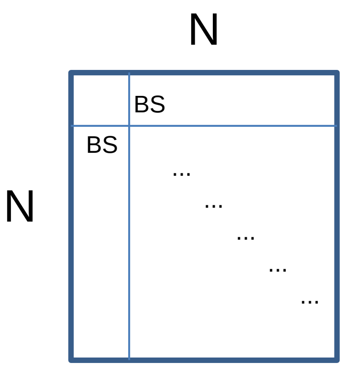



```
for (int j; j<N; j+=BS){

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

   #pragma oss 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];
}
```



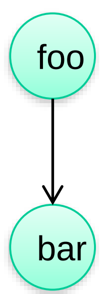
N = NB * BS

```
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 oss 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 oss task out (sentinel)
   foo (...);

#pragma oss 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

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 oss 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 oss 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 oss 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 oss 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 oss taskwait
}
```

OmpSs: syntax in Fortran

```
!$OSS TASK [IN (…)] [OUT(...)] [INOUT (…)] &
         [CONCURRENT (…)][COMMUTATIVE (…)] &
         [PRIORITY (…)][LABEL (…)]&
         [SHARED(…)][PRIVATE(…)][FIRSTPRIVATE(…)][DEFAULT (…)]&
         [UNTIED][IF(…)][FINAL(…)] &
         [DEVICE({SMP|CUDA|OPENCL|MPI})]&
         [NDRANGE (…)]&
         [IMPLEMENTS (function_name)]* &
         [SHMEM (…)]
{ code or function }
```

```
!$OSS TASKWAIT [ON(…)][NOFLUSH]
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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
!$oss task out (v(jj)) in (vx(jj), vy(jj), vz(jj)) firstprivate (jj) \
                                                          label(vmod)
    call v_mod(BSIZE, v(jj), vx(jj), vy(jj), vz(jj))
!$oss end task
enddo
...
!$oss taskwait
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

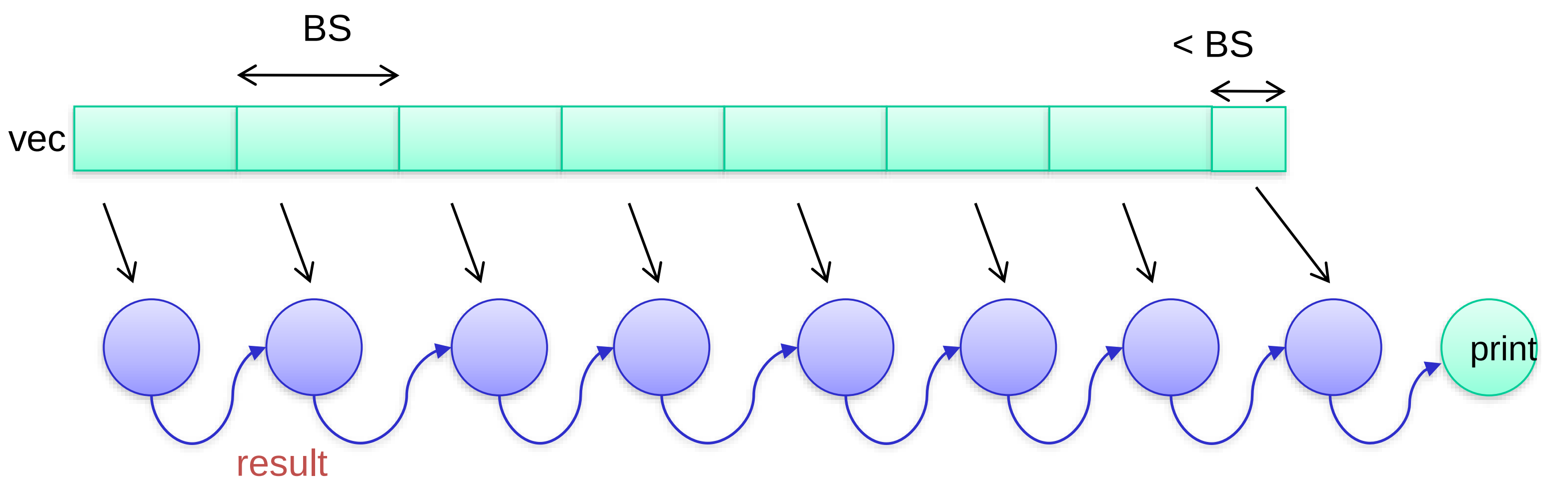
Concurrent

```
#pragma oss task concurrent (lvalue_expr)    // inoutset in OMP
```

- **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 oss atomic
#pragma oss critical
```

Serialized reduction pattern

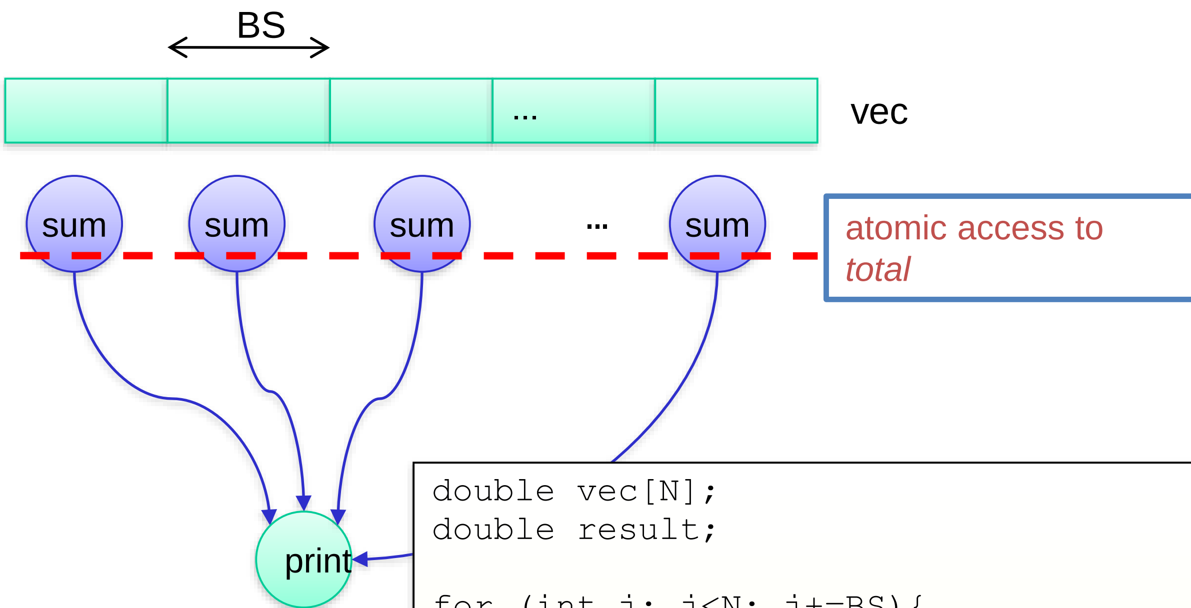



```
for (int j=0; j<N; j+=BS){

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

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

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

Serialization

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Concurrent




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

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

   #pragma oss 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 oss atomic
      result += local_result;
   }
}
#pragma oss task input (result)
  printf ("TOTAL is %d\n", result);
#pragma oss taskwait
```

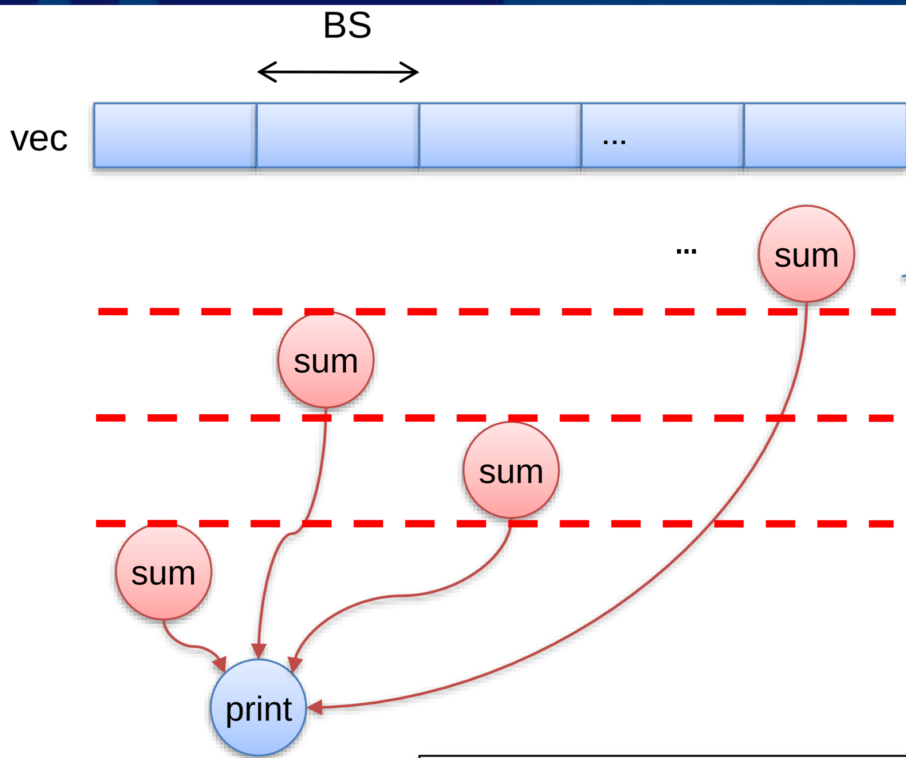
Commutative

```
#pragma oss task commutative (lvalue_expr)
                                   // mutexinoutset in OMP
```

- **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.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Commutative



Tasks executed out of order but not concurrently

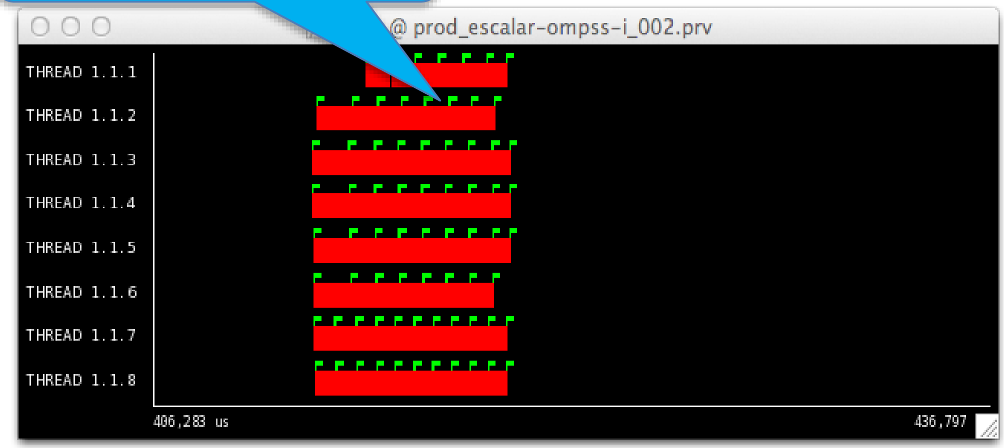
```
for (int j; j<N; j+=BS){

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

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

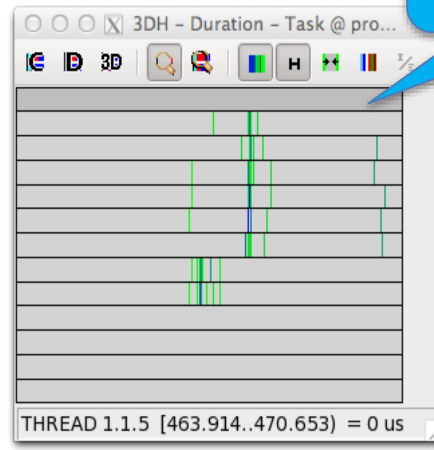
No mutual exclusion required

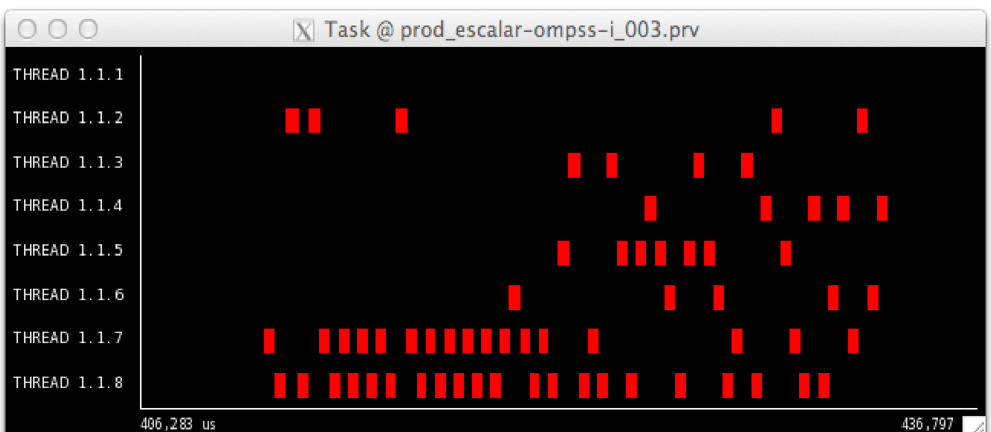
Differences between concurrent and commutative

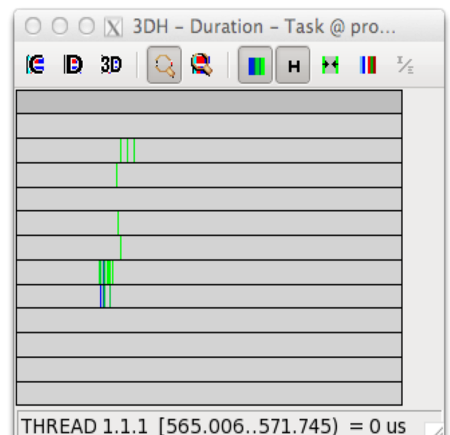
Tasks timeline: views at same time scale

Histogram of tasks duration: at same control scale









In this case, concurrent is more efficient

… but tasks have more duration and variability

Task nesting

Hierarchical task graph

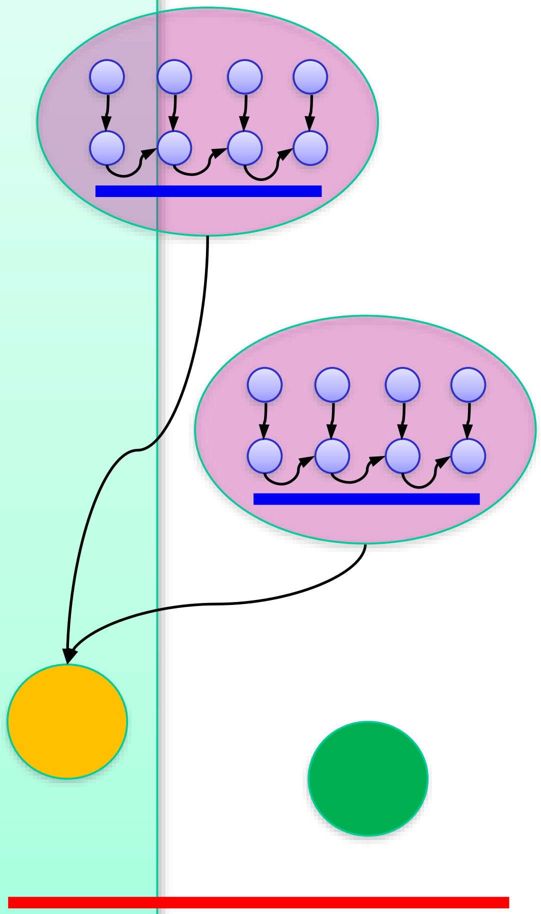
- Nesting
  - As in OpenMP
  - Tasks can generate tasks themselves
  - Everything said so far now applies within the context of a task whose sequential control flow now becomes the main in the local context

- 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

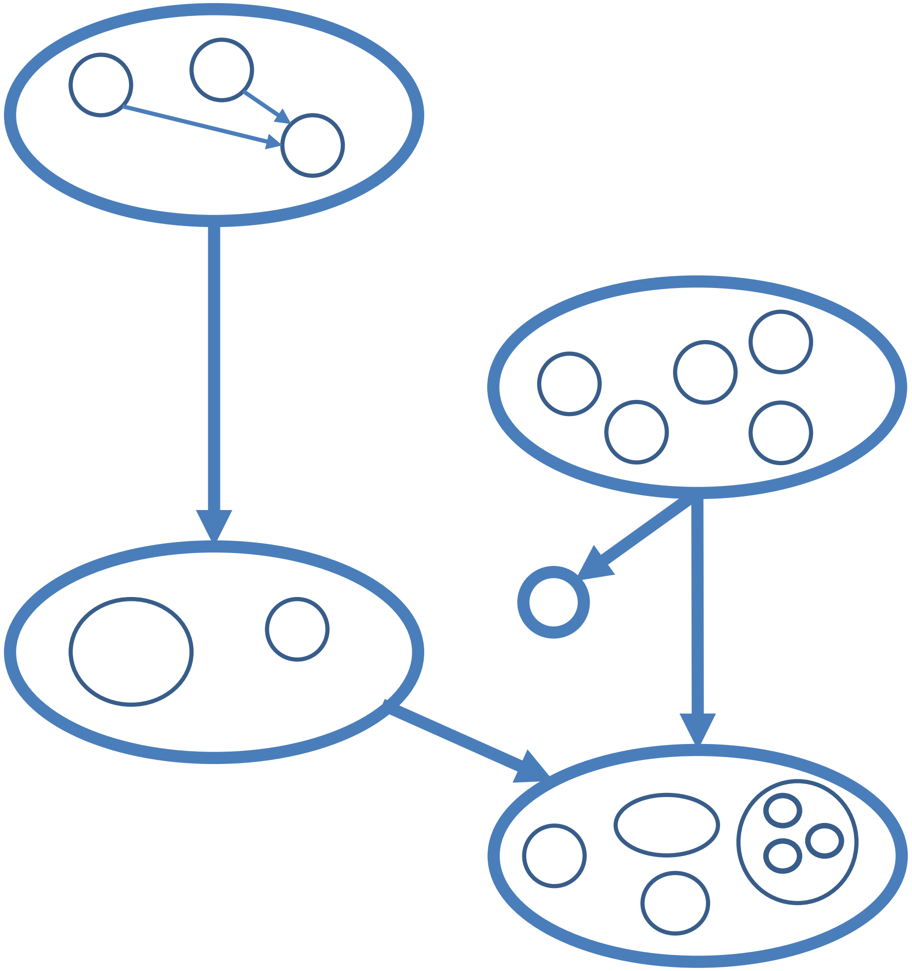
Nesting

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

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

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

   #pragma oss taskwait
}
```



Nesting

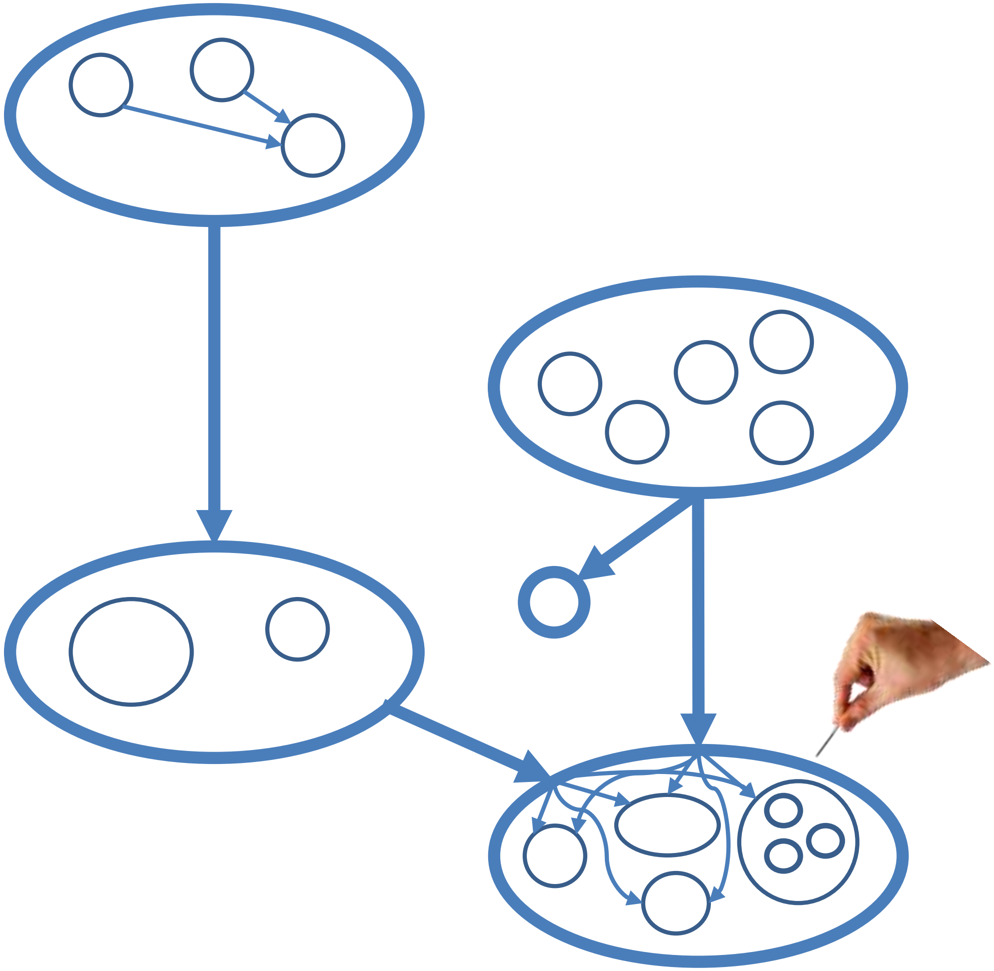
- Top down
  - Every level contributes
- Flattening dependence graph
  - Increase concurrency
  - Take out runtime overhead from critical path
- Granularity control
  - final clauses, runtime



J. M. Perez, et all, "Improving the Integration of Task Nesting and Dependencies in OpenMP" IDPS 2017

Nesting

- Top down
  - Every level contributes
- Flattening dependence graph
  - Increase concurrency
  - Take out runtime overhead from critical path
- Granularity control
  - final clauses, runtime

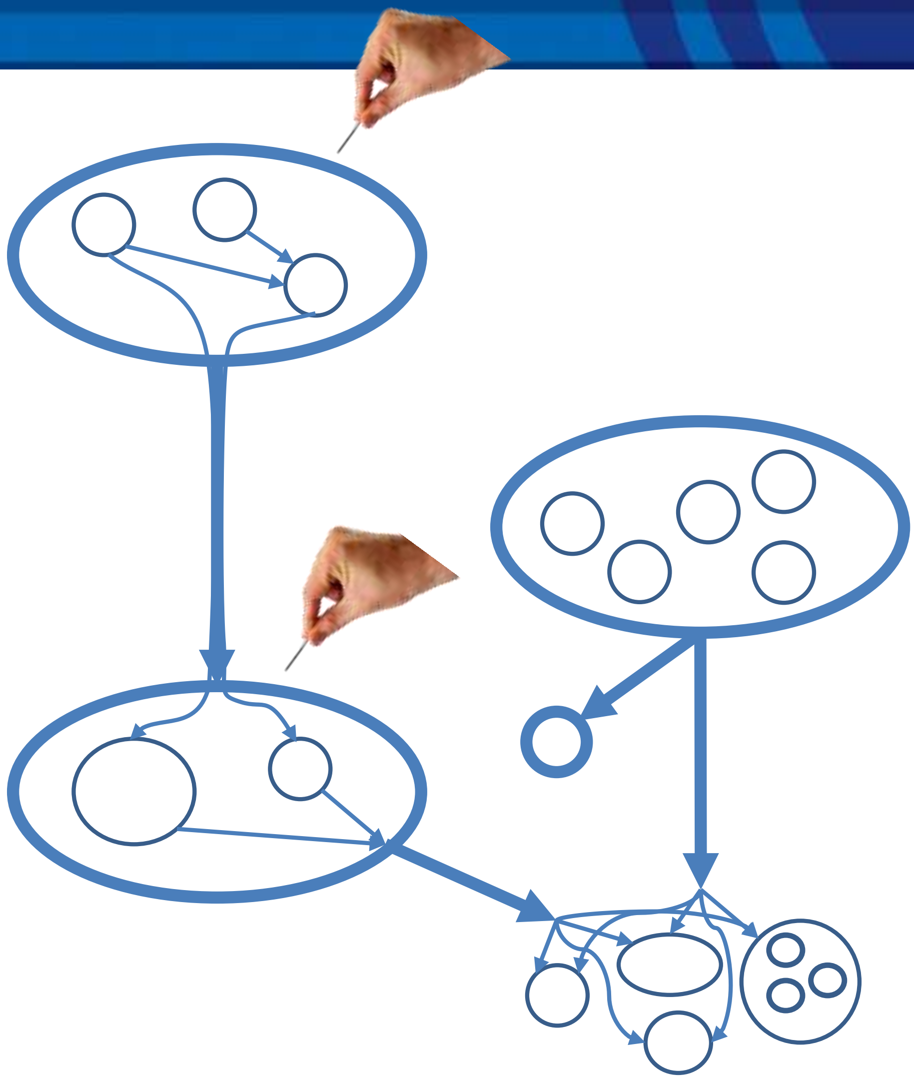


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

J. M. Perez, et all, "Improving the Integration of Task Nesting and Dependencies in OpenMP" IPDPS 2017

Nesting

- Top down
  - Every level contributes
- Flattening dependence graph
  - Increase concurrency
  - Take out runtime overhead from critical path
- Granularity control
  - final clauses, runtime



J. M. Perez, et all, "Improving the Integration of Task Nesting and Dependencies in OpenMP" IDPS 2017

OmpSs-2: Improved Nesting + Dependences

```
#pragma oss task weakout (a, b, c)
{
   #pragma oss task out(a)
   {…}
   #pragma oss task out(b)
   {…}
   #pragma oss task in(a, b) out(c)
   {…}
}
```

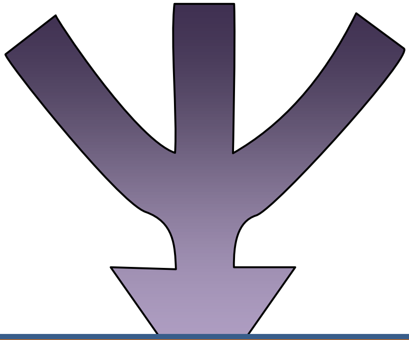
After finishing, release dependences incrementally

Connect Inner Dependencies Outwards

```
#pragma oss task weakin(a, c)
{
   #pragma oss task in(a)
   {…}
   #pragma oss task in(c)
   {…}
}
```

Do not wait for these dependences

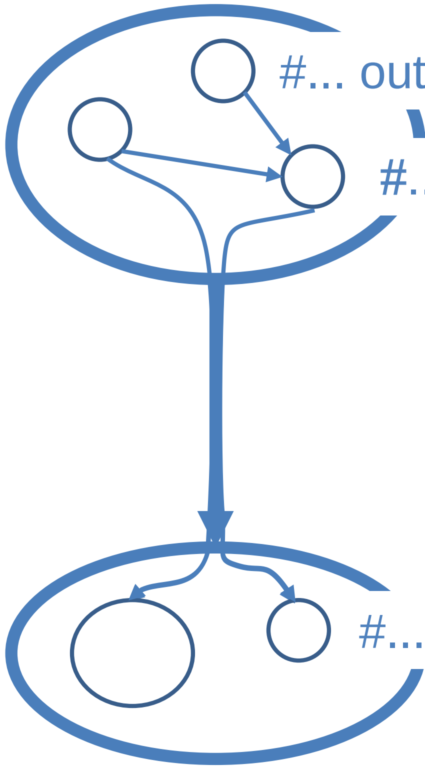
Wait only for the inner ones



Connect Outer Dependencies Inwards

OmpSs-2: Improved Nesting + Dependences

#pragma oss task **weak**out(a, b, c)



#pragma oss task **weak**in(a, c)

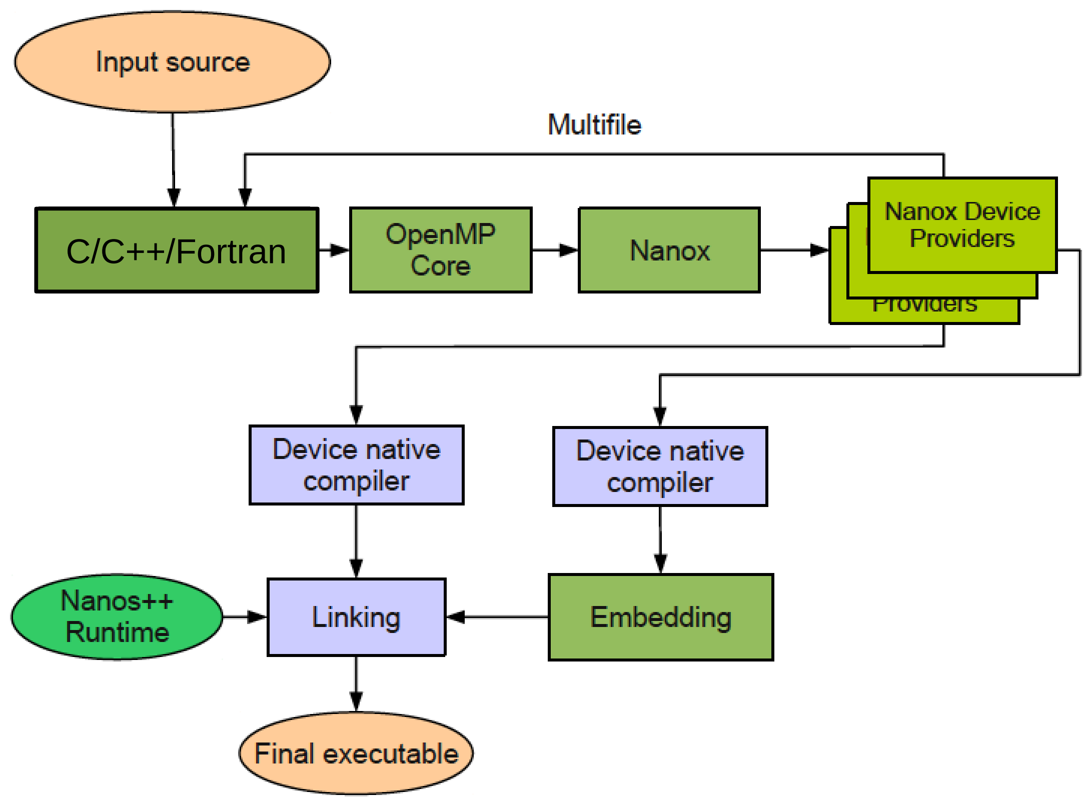
Our team

- Jesus Labarta
- Eduard Ayguade
- Rosa M. Badia
- Xavier Martorell
- Vicenç Bertran
- Antonio Peña
- Daniel Jiménez
- Carlos Álvarez
- Xavier Teruel
- Roger Ferrer
- Marta Garcia
- Victor Lopez
- Julian Morillo
- Kevin Sala
- Antoni Navarro
- Antonio Filgueras
- Daniel Peyrolón
- Miquel Vidal
- Orestis Korakitis
- Simon Garcia
- Judit Gimenez
- German Llort
- Harald Servat (Intel)
- Aymar Rodríguez
- Jaume Bosch
- Guray Ozen (Nvidia/PGI)
- Artem Cherkashin
- Marc Josep
- Marc Marí
- Ferran Pallarès
- Albert Navarro (Semidynamics)
- Alex Duran (Intel)
- Josep M. Perez
- Javier Bueno (Metempsy)
- Judit Planas (Lausanne)
- Guillermo Miranda (UPC)
- Florentino Sainz (BBVA)
- Omer Subasi
- Javier Arias
- Sergi Mateo (MongoDB)
- …

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



Compiling

- Compiling

```
frontend --ompss-2  -c bin.c
```

- Linking

```
frontend --ompss-2 -o bin bin.o
```

- where *frontend* can be:

mcc	C (gcc backend)
mcxx	C++
mfc	Fortran (gfortran backend)
imcc	C (icc backend)
imcxx	C++ (icpc backend)
imfc	Fortran (ifort backend)

Compiling

- Compatibility with OpenMP
  - --openmp-compatibility
  - Accepts OmpSs/OpenMP directives
    - #pragma omp …
  - Do not use in this course. Let the omp directives for the GCC compiler

  - [Useful for the taskloop support
    - From previous editions of the course
    - Now taskloop is already implemented in OmpSs-2]

Compiling

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

-k	Keep intermediate files
--debug	Use Nanos++ debug 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    # will use all cores/hwthreads available
```

  - No LD_LIBRARY_PATH or LD_PRELOAD needed

- Number of threads can be adjusted with “taskset”

```
> taskset -c 20-23 ./bin
```

- In the hands-on

```
> sbatch run-once.sh    # and slurm sets the cpu mask
```

Nanos6 helper

- NANOS6 utility to
  - List compiler and compilation options used for the runtime
    - nanos6-info --runtime-compiler
    - nanos6-info --runtime-compiler-flags
  - List runtime details:

```
nanos6-info --runtime-details
```

Runtime path /apps/PM/ompss-2/2020.11.1/lib/libnanos6-optimized.so.0.0.0
Runtime Version 2.5.1 2020-12-22 18:47:30 +0100 443f6d83
Runtime Branch
Runtime Compiler Version g++ (GCC) 6.4.0
Runtime Compiler Flags -DNDEBUG -std=gnu++11 -Wall -Wextra -Wdisabled-optimization -Wshadow -fvisibility=hidden -O3 -flto
Initial CPU List 0-47
NUMA Node 0 CPU List 0-23
NUMA Node 1 CPU List 24-47
Scheduler priority
Dependency Implementation linear-regions-fragmented
Threading Model pthreads

Tracing

- Compile and link normally

```
mcc --ompss-2 -c bin.c
mcc -o bin --ompss-2 bin.o
```

- When executing specify which instrumentation module to use:

```
NANOS6_CONFIG_FILE=extrae.xml
NANOS6_CONFIG_OVERRIDE="version.instrumentation=extrae" ./bin
```

- Will generate trace files in executing directory
  - 3 files: prv, pcf, rows
  - Use paraver to analyze
- In the hands-on
  - Add ./trace.sh to run-once.sh to enable tracing

  > ./trace.sh $PROGRAM … args …

```
> sbatch run-once.sh
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Nanos6 configuration file

- Installed in share/nanos6/scripts/nanos6.toml
  - [scheduler]   fifo, lifo…
  - [version] debug, dependencies (discrete, regions), instrument
  - [cpumanager] policy (default, idle, busy, lewi, greedy)
  - [throttle] enabled = false
    - tasks = 5000000
    - pressure = 70   #%
    - max_memory = "0"   # half of the system memory
  - [numa]
  - [dlb]
  - …

Nanos++ scheduling policies

- Use of schedulers:

  – fifo   # default
  – lifo

  – immediate_successor   # true by default

- Changing the policy
  – NANOS6_CONFIG_OVERRIDE="scheduler.policy = lifo"

NANOS6 versions

- NANOS6 has runtime versions with different characteristics
  - extrae
  - ctf
  - graph # generates the graph step by step
  - lint
  - stats
  - verbose

- Example:

```
> NANOS6_CONFIG_OVERRIDE="instrument = extrae"  \
      taskset   -c 0-23    ./bin
```

Task graph generation

- Compile and link normally

```
mcc --ompss-2 -c bin.c
mcc -o bin --ompss-2 bin.o
```

- When executing specify graph as instrumentation package:

```
NANOS6_CONFIG_OVERRIDE=\
                "version.instrument=graph"
./bin
```

- Will generate one dot file per graph step
  - graph-XXX-YYY.sh # script to generate PDF files
  - Graph-XXX-YYY-components # directory with the graph steps

System installation

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

- OmpSs components downloadable from
  - Nanos++ runtime
    - https://github.com/bsc-pm/nanos6
    - Installation: configure & make
  - Mercurium compiler
    - https://pm.bsc.es/ompss-downloads
    - Installation: configure & make

- Installation guide:
  - https://pm.bsc.es/ompss-2-docs/user-guide/build/index.html

Reporting problems

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

PATC Courses

- MPI + OpenMP & OmpSs
  - October, 2022 [physical – at this point]
- Distributed programming with COMPSs
  - January 25-26, 2022 [physical/online, to be decided]
- Performance analysis tools
  - March 16-17, 2022 [physical – at this point]
- MPI + OmpSs, Heterogeneous prog. OpenACC/FPGAs
  - March 23-25, 2022 [physical – at this point]

- Check https://www.bsc.es/education/training/patc-courses

- Free of charge

THANKS

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Visualizing timelines

Visualizing Paraver tracefiles

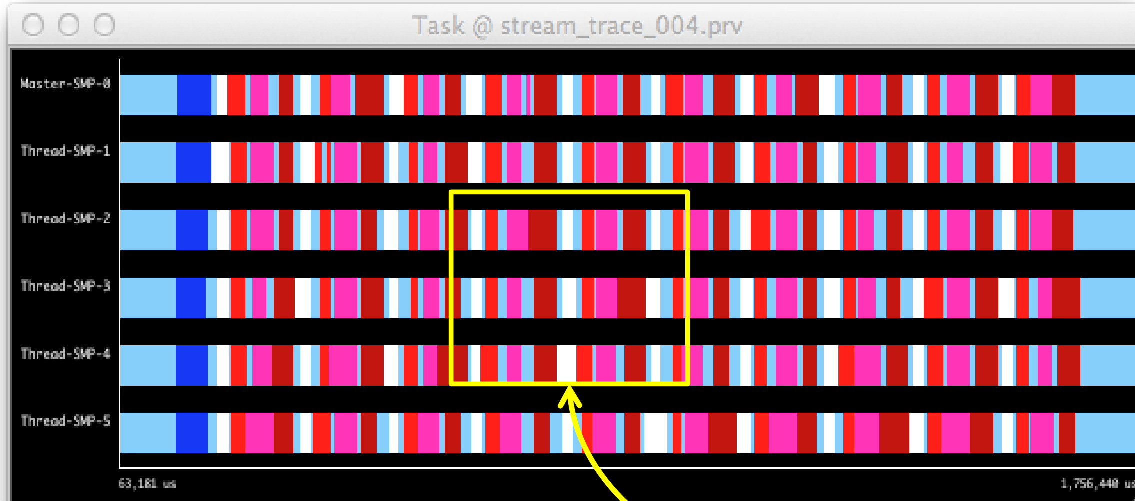
```
$ mcc --ompss -o prog -c prog.c
$ export EXTRAE_CONFIG_FILE=extrae.xml
$ NANOS6_CONFIG_OVERRIDE=\
        "version.instrumentation=extrae" taskset -c 0-7  ./prog
```

- 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

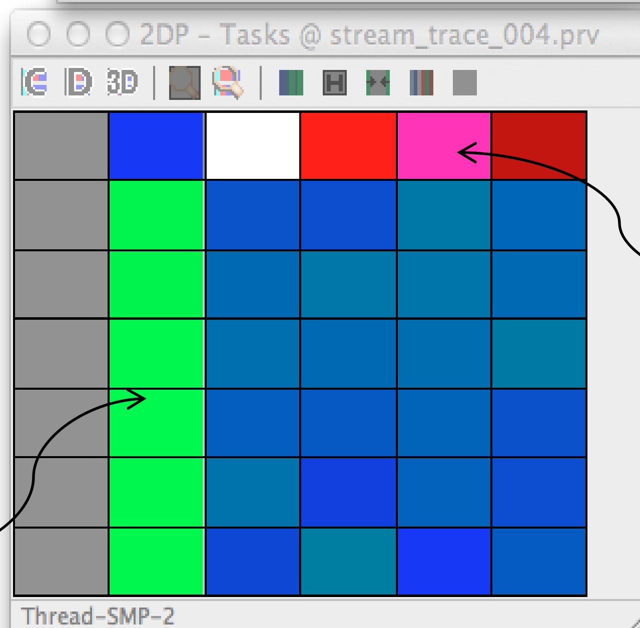
Tasks' profile

- 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

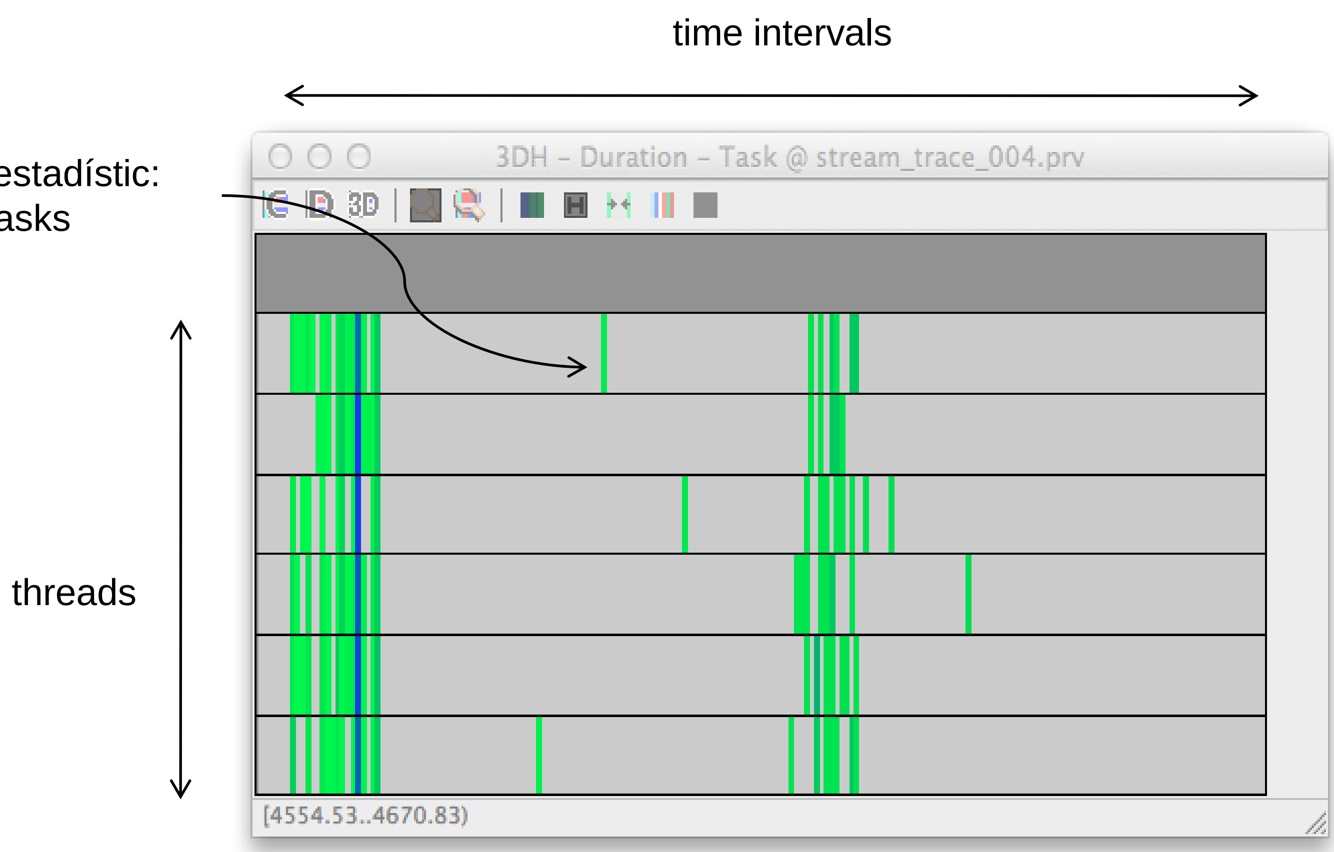
threads

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

Tasks duration histogram

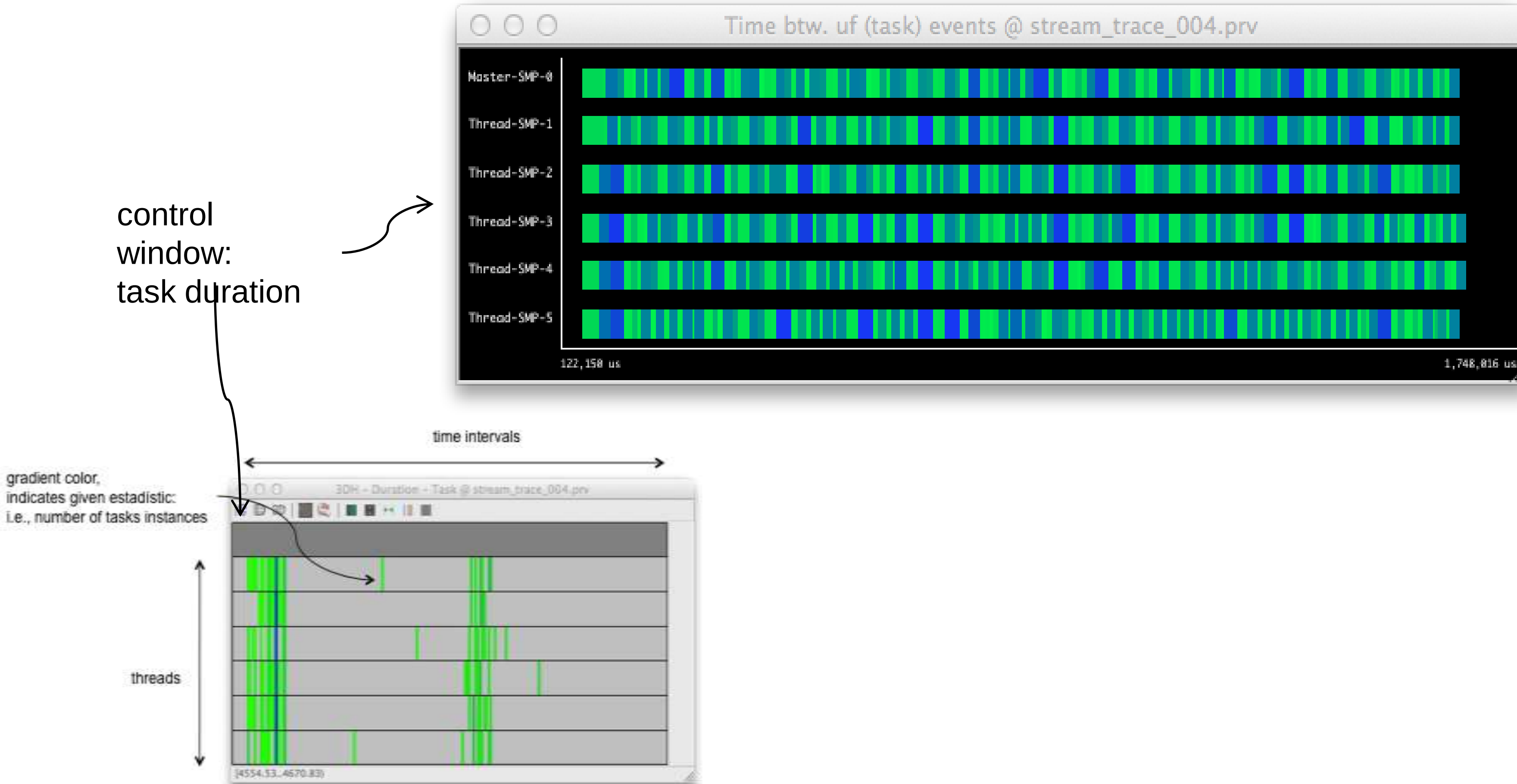
- 3dh_duration_task.cfg

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



Tasks duration histogram

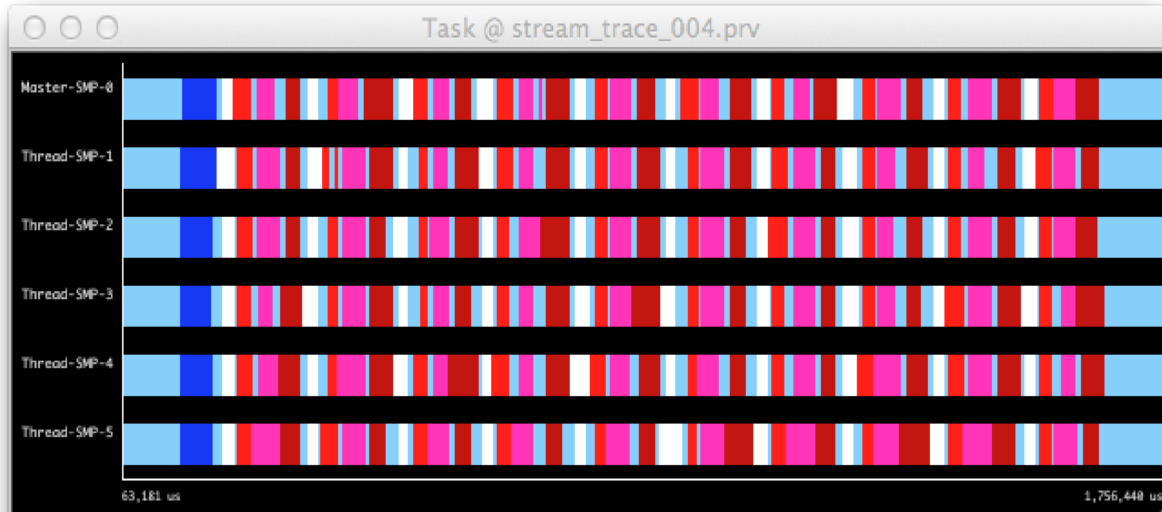
- 3dh_duration_task.cfg



Tasks duration histogram

3dh_duration_task.cfg

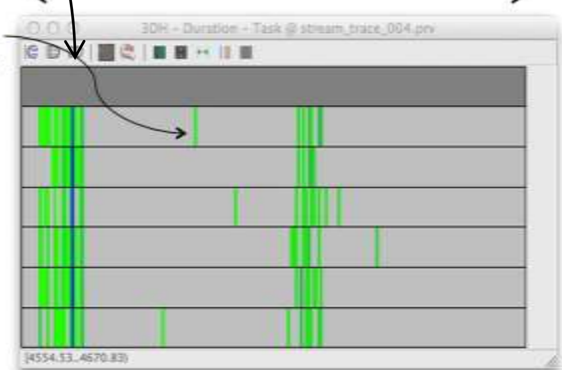
3D window:
task type



time intervals

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

threads



Tasks duration histogram

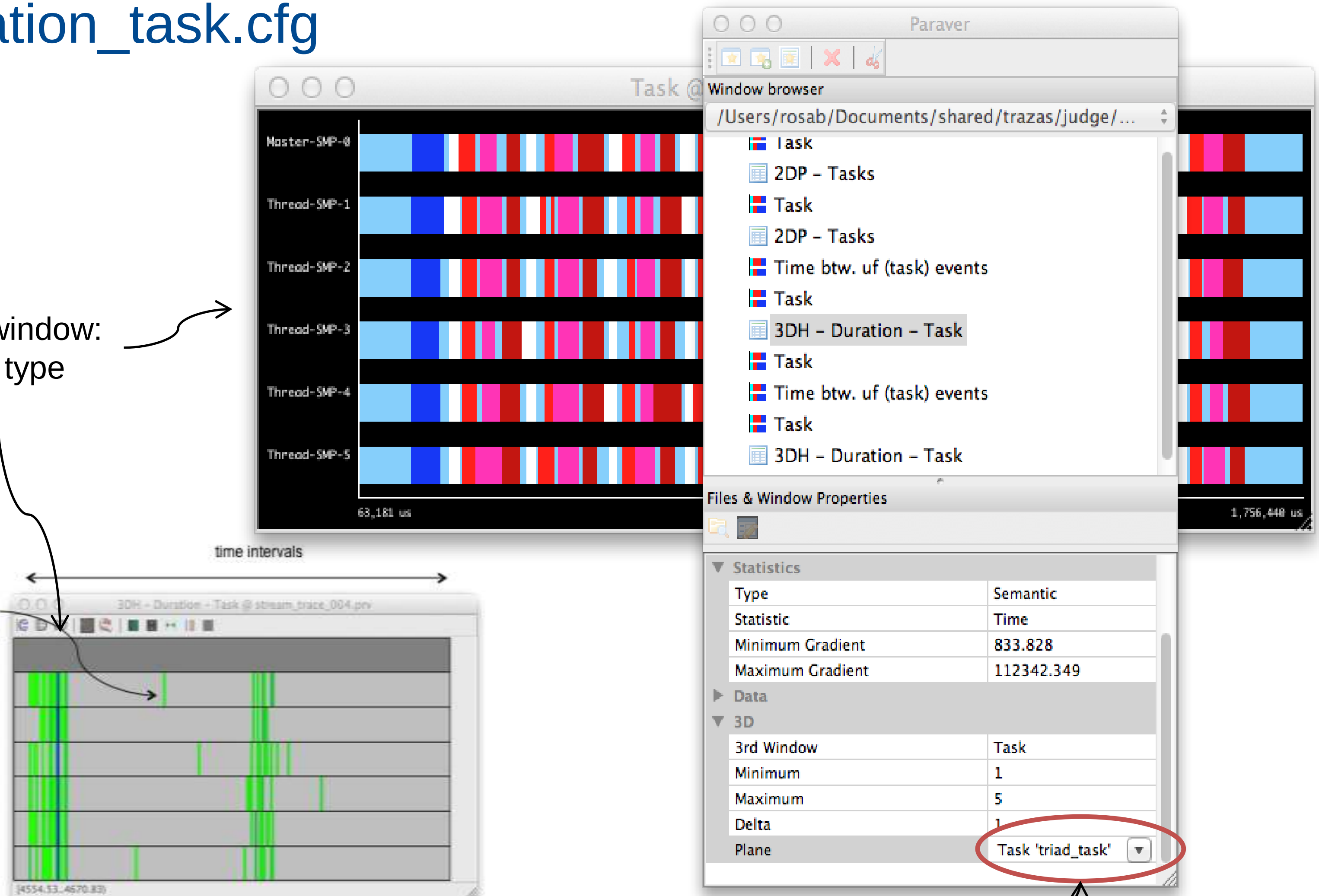
- 3dh_duration_task.cfg

3D window:
task type

time intervals

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

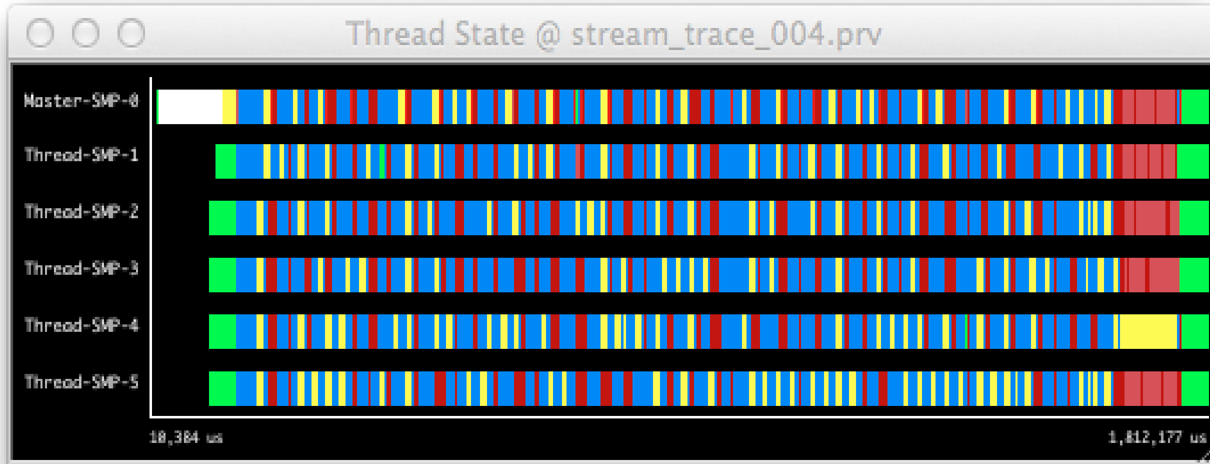
threads



chooser:
task type

Threads state profile

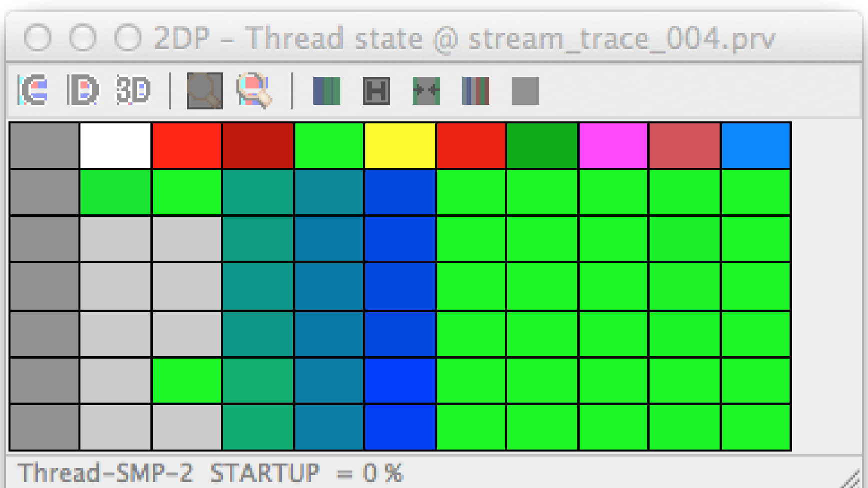
- 2dp_threads_state.cfg



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

runtime state

threads



Tasks & data scoping

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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 oss task firstprivate (i)
for ( ; i< 4; i++) X[i]=i;
#pragma oss 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 oss task private(i)shared(X)
for (i=0; i< 100; i++) X[i]=i;
#pragma oss taskwait

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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 oss task firstprivate (i)
for ( ; i< 4; i++) X[i]=i;
#pragma oss 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 oss task private(i)shared(X)
for (i=0; i< 100; i++) X[i]=i;
#pragma oss taskwait

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

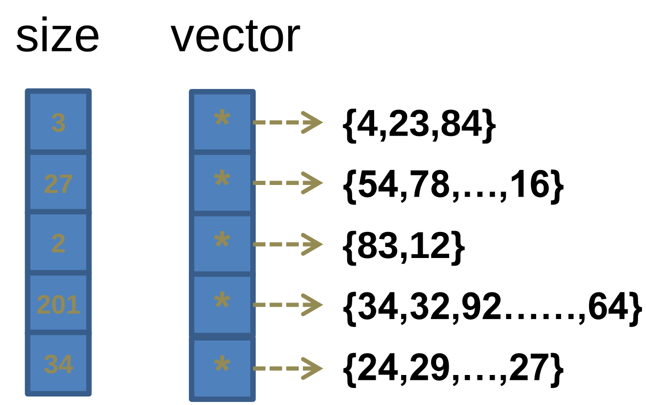
Additional directives

Tasks: the if clause

#pragma oss 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 oss task if ( size[i] > MIN_SIZE )
   compute(vector[i], size[i]) ;
 }
}
```



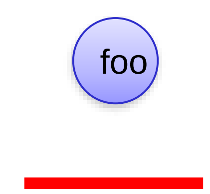
Outlined directives

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

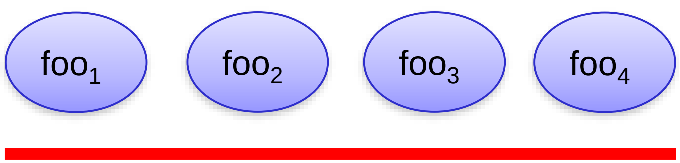
```
#pragma  oss   task
void   foo (int Y[size], int size) {
   for (int j=0; j<size; j++) Y[j]= j;
}

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

int main()
{
   int X[100];

   foo (X, 100) ;
  #pragma oss 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

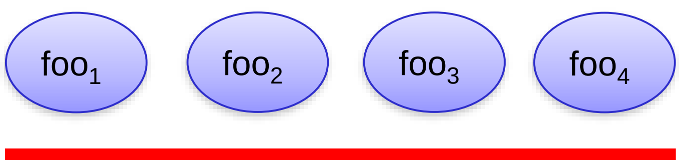



```
#pragma  oss   task
void   foo (int Y[size], int size, int a)
{
   for (int j=0; j<size; j++) Y[j]= a++;
}
```

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

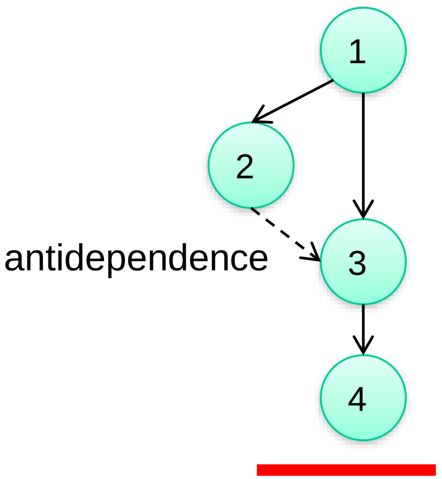
```
int Y[6]={1,2,3,4,5,6};

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 oss 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 oss task out(*px)
void set (int *px, int v) {*px = v;}

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

#pragma oss 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 oss taskwait
```

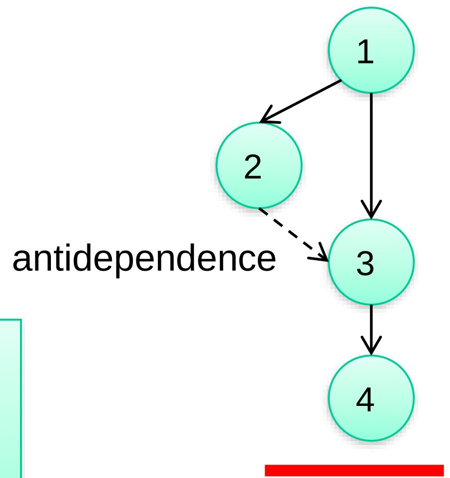


Defining dependences for outlined tasks

- 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 oss task out(*px)
void set (int *px, int v) {*px = v;}

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

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

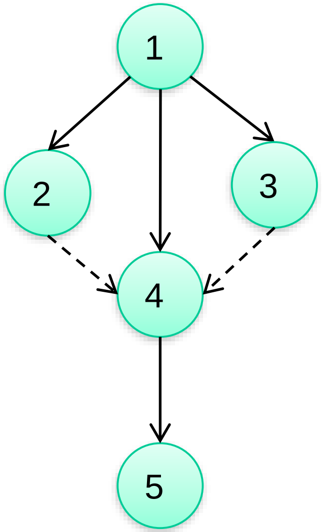
```
set(&x,5);        //1
do_print(x);    //2
x = incr(x);          //3
do_print(x);    //4
#pragma oss taskwait
```



Mixing inlined and outlined tasks

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

int main()
{
int x;

   x=3;

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

non-taskified:
executed
sequentially

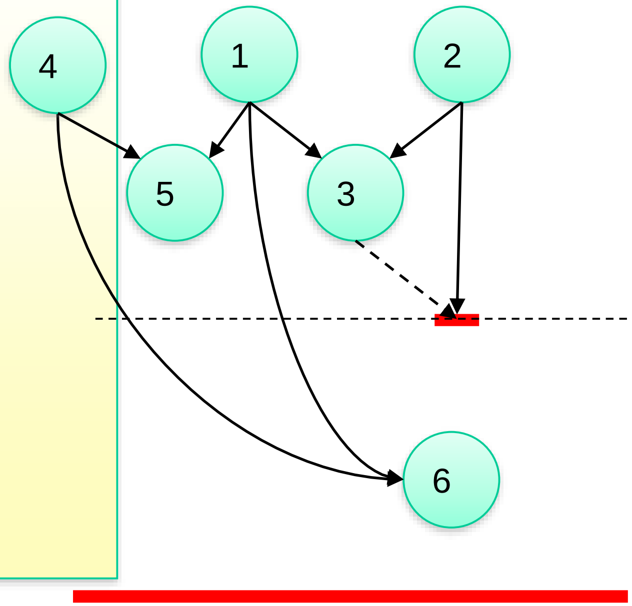


Partial control flow synchronization

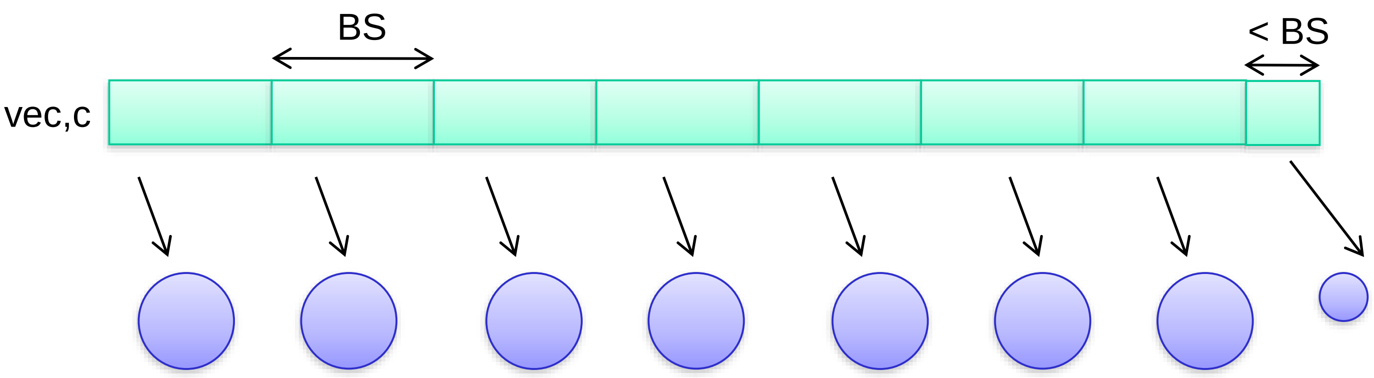
#pragma oss taskwait on (lvalue_expr-list)

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

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

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

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



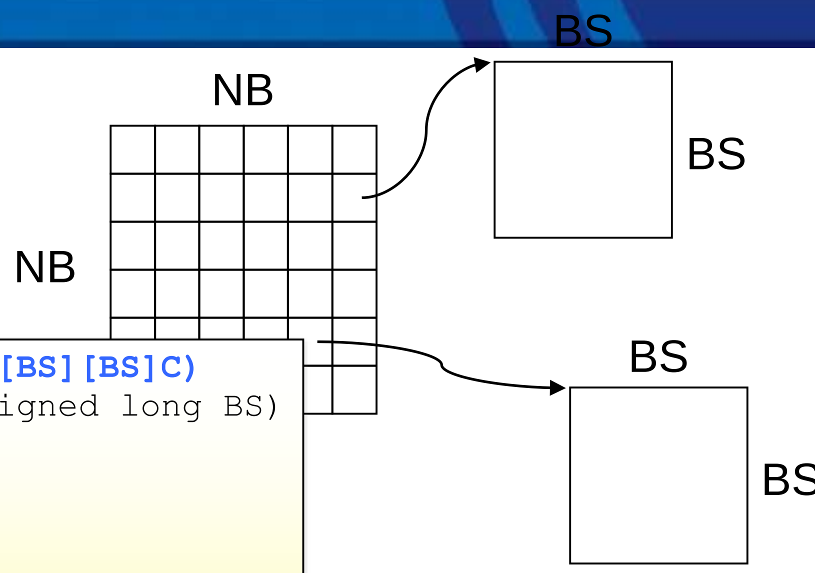
Array sections in outlined tasks




```
#pragma oss 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

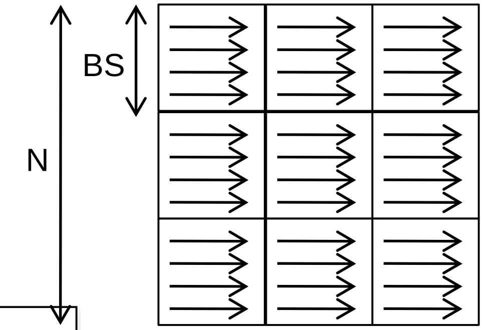



```
#pragma oss task in([BS][BS]A, [BS][BS]B) inout([BS][BS]C)
void matmul(double *A, double *B, double *C, unsigned long BS)
{
  int i, j, k;

  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

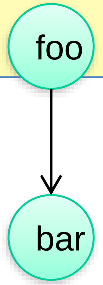



```
#pragma oss task in([BS][BS]A, [BS][BS]B) inout([BS][BS]C)
void matmul(double  *A, double *B, double *C, unsigned long BS)
{
  int i, j, k;

  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 oss task out (*sentinel)
void foo ( .... , int *sentinel){
   ...
}

#pragma oss 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 oss 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.

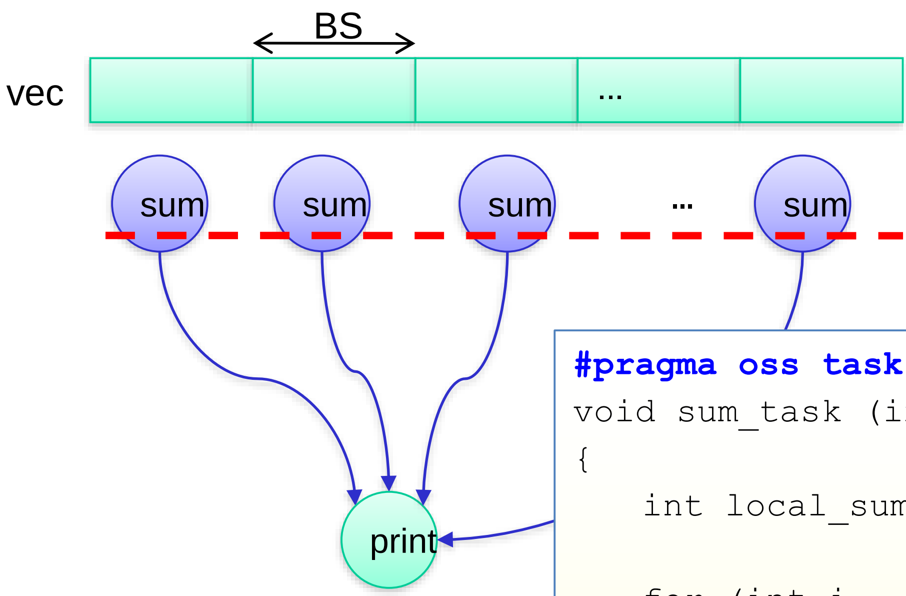
Does NOT provide actual information of data access pattern. (see copy clauses)

Fortran

- Pragmas outlined

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

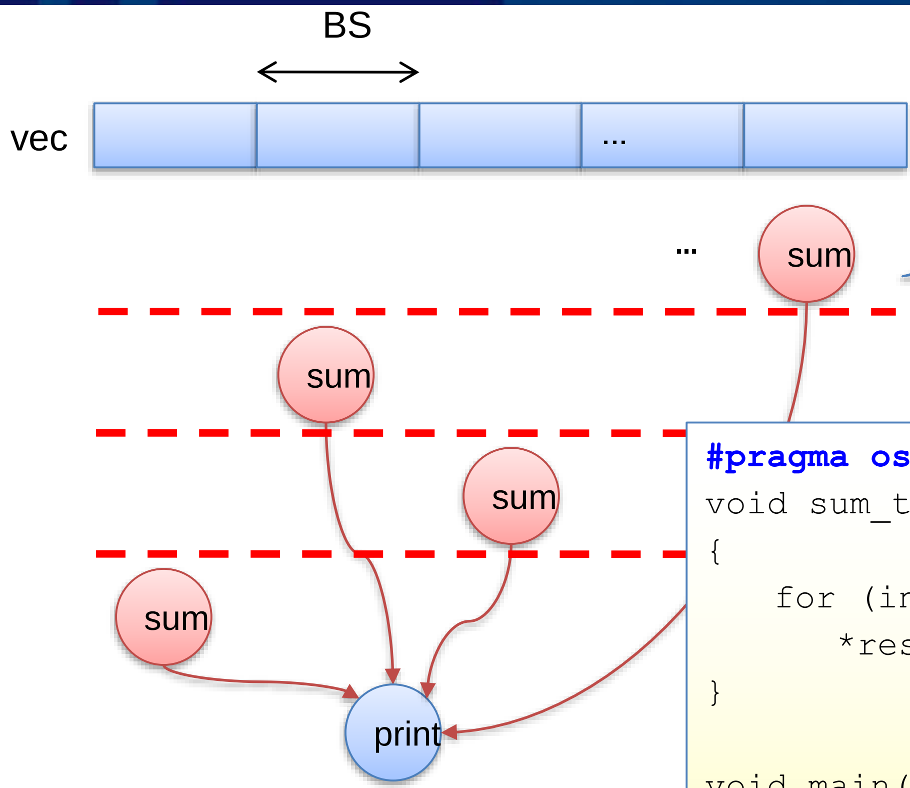
Concurrent in outlined tasks



atomic access to *total*

```
#pragma oss 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 oss atomic
    *result += local_sum;
}


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

Commutative in outlined tasks



Tasks executed out of order but not concurrently

```
#pragma oss 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 oss task in (total)
   printf ("TOTAL is %d\n", result);
   #pragma oss taskwait
}
```

No mutual access required