Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming with OmpSs

Introduction to the OmpSs programming model

Computer Sciences Research Dept.
BSC

Agenda

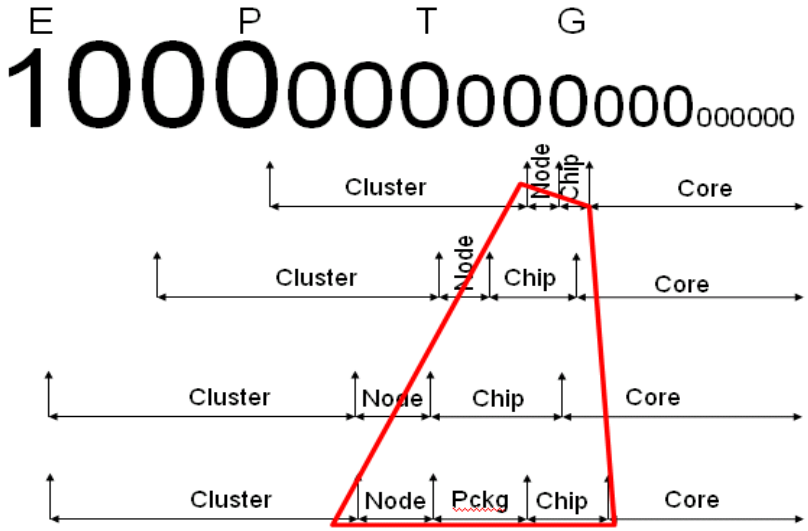
- The StarSs family of task-based programming models
- OmpSs syntax and examples
  - Inlined tasks
  - Outlined tasks
- Infrastructure
- Conclusion

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

Challenges on the way to Exascale

- Efficiency (…, power, …)
- Variability
- Memory
- Faults
- Scale (…,concurrency, strong scaling,…)
- Complexity (…Hierarchy /Heterogeneity,…)

"To kill flies with canons"
Spanish saying



J. Labarta, et al, "BSC Vision towards Exascale"
IJHPCA vol 23, n. 4 Nov 2009

"The butterfly effect

Supercomputer Development

Application
Algorithm
Progr. Model
Run time
Architecture

Is any of them more important than the others?

The sword to cut the “multicore” Gordian Knot

The StarSs family of programming models

- Key concept
  - Sequential **task based** program on **single address/name space**
    - Discipline
    - Blocked to achieve sufficient granularity → performance
  - **Directionality annotations**

- Executed Parallel
  - Automatic run time computation of dependencies

- Targeting productivity
  - “single” program → any platform

StarSs tasks

- Task
  - Computation unit. Amount of work (granularity) may vary in a wide range (microseconds to milliseconds or even seconds), may depend on input arguments,…
  - Once started can execute to completion independent of other tasks
- States:
  - **Instantiated**: when task is created by main program. 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/Executing**: 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.

StarSs: a sequential program …

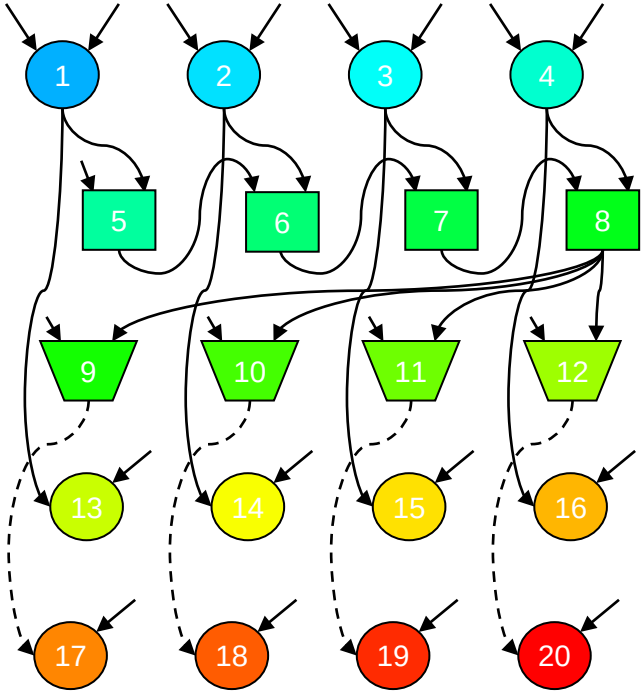
```
void vadd3 (float A[BS], float B[BS],
            float C[BS]);

void scale_add (float *sum, float A[BS],
                float B[BS]);

void accum (float A[BS], float *sum);
```

```
for (i=0; i<N; i+=BS)              // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)             // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)             // B=sum*A
   scale_add (&sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)             // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)             // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

StarSs: … taskified …

```
#pragma omp task input(A, B) output(C)
void vadd3 (float A[BS], float B[BS],
            float C[BS]);
#pragma omp task input(*sum, A) inout(B)
void scale_add (float *sum, float A[BS],
                float B[BS]);
#pragma omp task input(A) inout(*sum)
void accum (float A[BS], float *sum);
```

```
for (i=0; i<N; i+=BS)              // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)             // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)             // B=sum*A
   scale_add (&sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)             // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)             // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

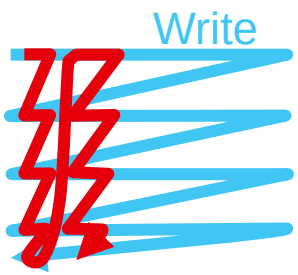


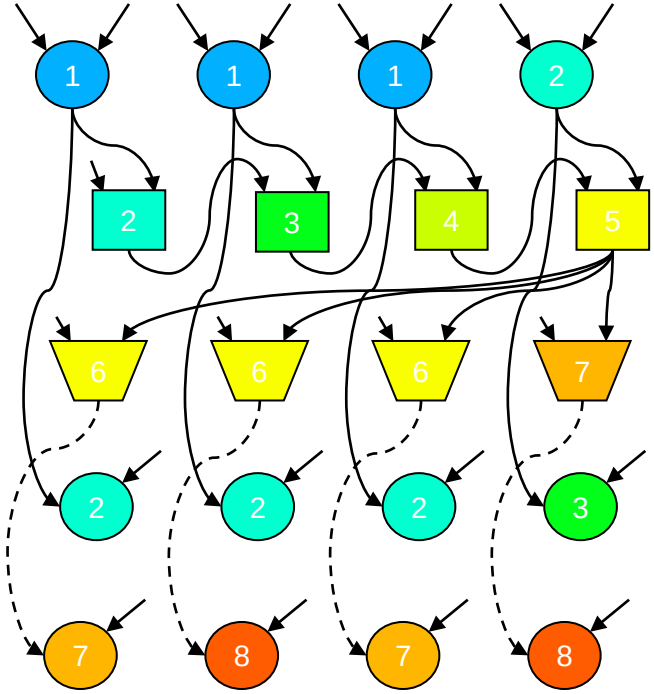
StarSs: … and executed in a data-flow model

```
#pragma omp task input(A, B) output(C)
void vadd3 (float A[BS], float B[BS],
            float C[BS]);
#pragma omp task input(*sum, A) inout(B)
void scale_add (float *sum, float A[BS],
                float B[BS]);
#pragma omp task input(A) inout(*sum)
void accum (float A[BS], float *sum);
```

```
for (i=0; i<N; i+=BS)              // C=A+B
   vadd3 ( &A[i], &B[i], &C[i]);
...
for (i=0; i<N; i+=BS)             // sum(C[i])
   accum (&C[i], &sum);
...
for (i=0; i<N; i+=BS)             // B=sum*A
   scale_add (&sum, &E[i], &B[i]);
...
for (i=0; i<N; i+=BS)             // A=C+D
   vadd3 (&C[i], &D[i], &A[i]);
...
for (i=0; i<N; i+=BS)             // E=G+F
   vadd3 (&G[i], &F[i], &E[i]);
```

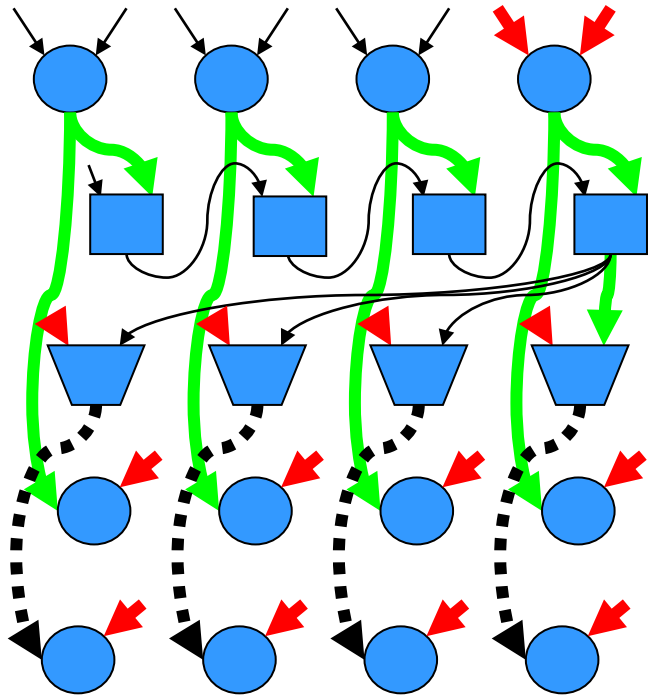
Decouple
how we write
form
how it is executed





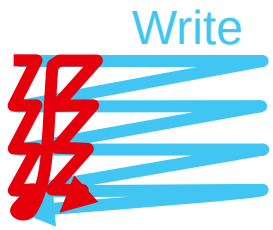
StarSs: the potential of data access information

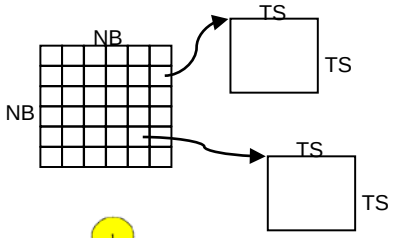
- **Flat global address space seen by programmer**
- Flexibility to dynamically traverse dataflow graph "optimizing"
  - Concurrency. Critical path
  - Memory access: data transfers performed by run time
- Opportunities for runtime to
  - Prefetch
  - Reuse
  - Eliminate antidependences (rename)
  - Replication management
    - Coherency/consistency handled by the runtime

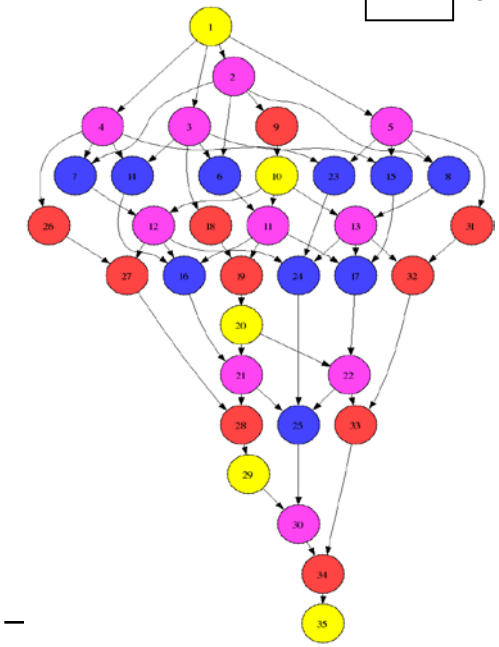


StarSs: data-flow execution of sequential programs

Decouple
how we write
form
how it is executed



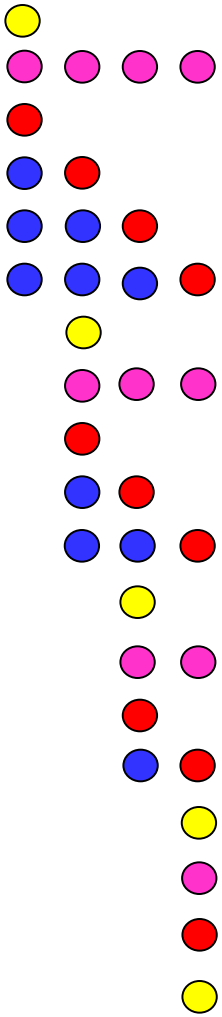




```
void Cholesky( float *A ) {
   int i, j, k;
   for (k=0; k<NT; k++) {
      spotrf (A[k*NT+k]) ;
      for (i=k+1; i<NT; i++)
         strsm (A[k*NT+k], A[k*NT+i]);
      // update trailing submatrix
      for (i=k+1; i<NT; i++) {
         for (j=k+1; j<i; j++)
            sgemm( A[k*NT+i], A[k*NT+j],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

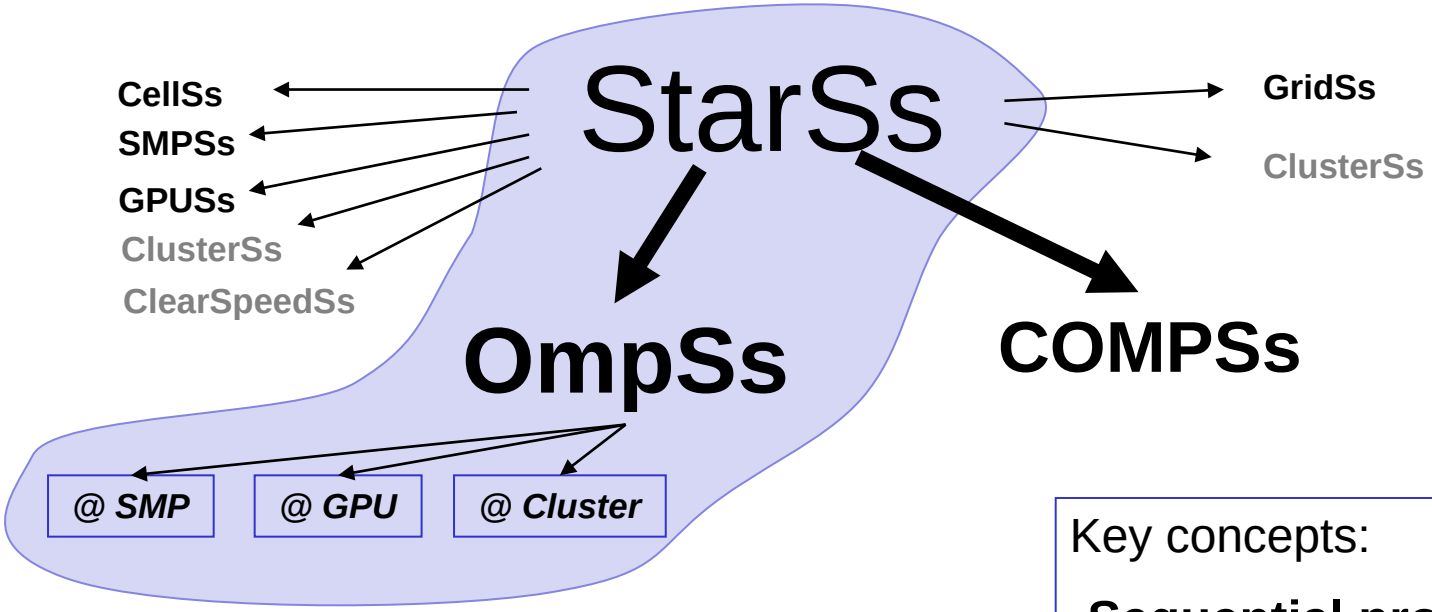
```
#pragma omp task inout ([TS][TS]A)
void spotrf (float *A);
#pragma omp task input ([TS][TS]T) inout ([TS][TS]B)
void strsm (float *T, float *B);
#pragma omp task input ([TS][TS]A,[TS][TS]B) inout ([TS][TS]C )
void sgemm (float *A, float *B, float *C);
#pragma omp task input ([TS][TS]A) inout ([TS][TS]C)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

StarSs vs OpenMP -- Dependences vs. fork-join



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

The StarSs family of programming models



Key concepts:

- **Sequential program**
- **Directionality annotations on tasks arguments**

- Task based. Asynchrony, data-flow.
- Simple linear address space
- Malleable
- Nicely hybridizes (MPI/StarSs)
- Natural support for heterogeneity

Key objectives:

- **Same program any platform**
- **Productive / Incremental programming**

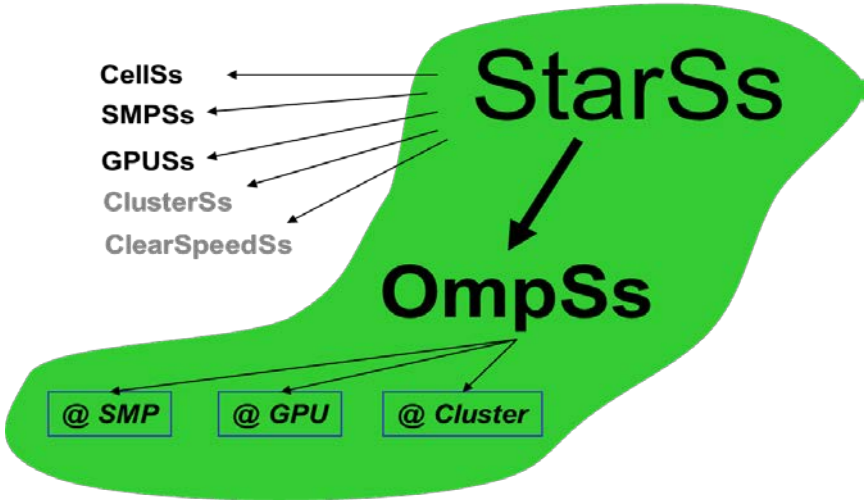
StarSs: history/versions

Basic SMPSs

must provide directionality ∀argument
Contiguous, non partially overlapped
Renaming
Several schedulers (priority, locality,…)
No nesting
C/Fortran
MPI/SMPSs optims.

SMPSs regions

C, No Fortran
must provide directionality ∀argument
ovelaping &strided
Reshaping strided accesses
Priority and locality aware scheduling

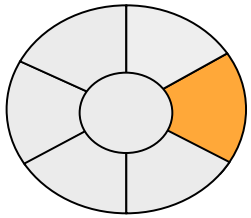


OmpSs

C, C++, Fortran
OpenMP compatibility (~)
Contiguous and strided args.
Separate dependences/transfers
Inlined/outlined pragmas
Nesting
Heterogeneity: SMP/GPU/Cluster
No renaming,
Several schedulers: "Simple" locality aware sched,…

Evolving research since 2005

OmpSs



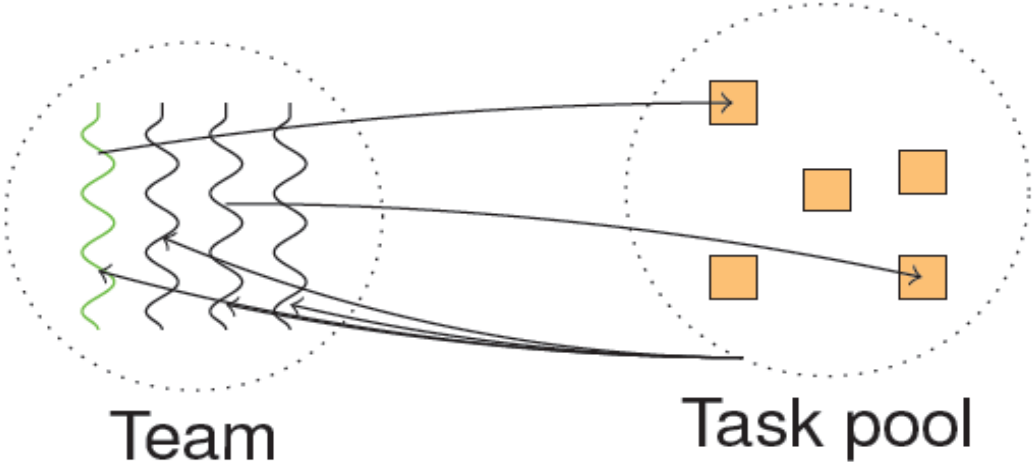
OmpSs = OpenMP + StarSs extensions

- OmpSs = OpenMP + StarSs extensions:
  - Different execution model
  - Extended memory model
  - Extensions for point-to-point inter-task synchronizations
    - data dependencies
  - Extensions for heterogeneity
  - Other minor extensions

- OmpSs = StarSs relaxations
  - Directionality for only some task arguments, incomplete specification, concurrent clause,…

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



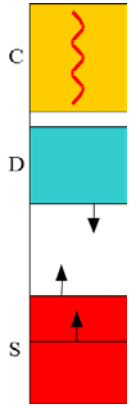
Memory Model

- 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

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,…)

OmpSs: a reasonable merge/evolution of OpenMP

blue already in OpenMP
red extensions based on StarSs

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

Support for multiple implementations of a task

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

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

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

```
#pragma omp task [ input (...)] [ output (...)] [ inout (...)] [ concurrent (...)]

{code block or function}
```

To compute dependences

To allow concurrent execution of commutative tasks

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

Master waits for sons or specific data availability

Relax consistency to main program

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

Inlined task instantiation and wait

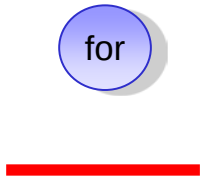
#pragma omp task

- Instantiates a task and control flow continues

#pragma omp taskwait

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

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

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

        ...
}
```



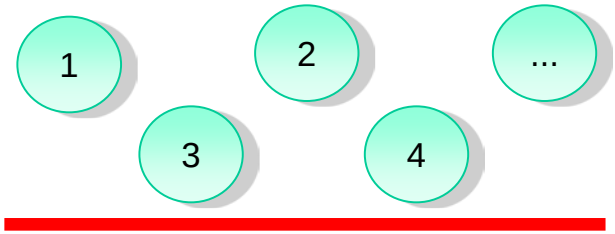
Inlined task instantiation and wait

#pragma omp task

- Instantiates a task and control flow continues

#pragma omp taskwait

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




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

#pragma omp taskwait
}
```

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

Data scope

- The data accessed by a task may be:
  - **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

- OmpSs scoping rules == OpenMP scoping rules

Inlined tasks data scope

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

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

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

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

Inlined tasks data scope

- Variables for which no clause appears on the pragma:
  - Arrays declared global are shared by default
  - For the rest a capture value at task instantiation time semantic is followed
    - Scalars and local arrays the value is captured to initialize a private copy of the task. (Equivalent to the firstprivate clause in OpenMP)
    - 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(100*sizeof(int));

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

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

Inlined tasks data scope

- Data scope (… as in OpenMP)
  - Variables can be explicitly declared as :
    - Shared
    - Private
    - Firstprivate
- Do declare in case of doubt

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

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

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

Inlined tasks data scope

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

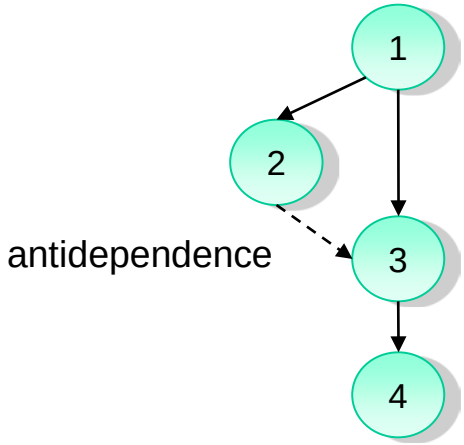
```
int X[4]={1,2,3,4};

int main( )
{

        int i=2;
        #pragma omp task
firstprivate (i)
        for ( ; i< 4; i++) X[i]=i;

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

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

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

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

Defining dependences for inlined tasks

- Clauses that express data direction:
  - Input, output, 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 reference it.
- Dependences computed at runtime taking into account these clauses
  - Flow, anti, output

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

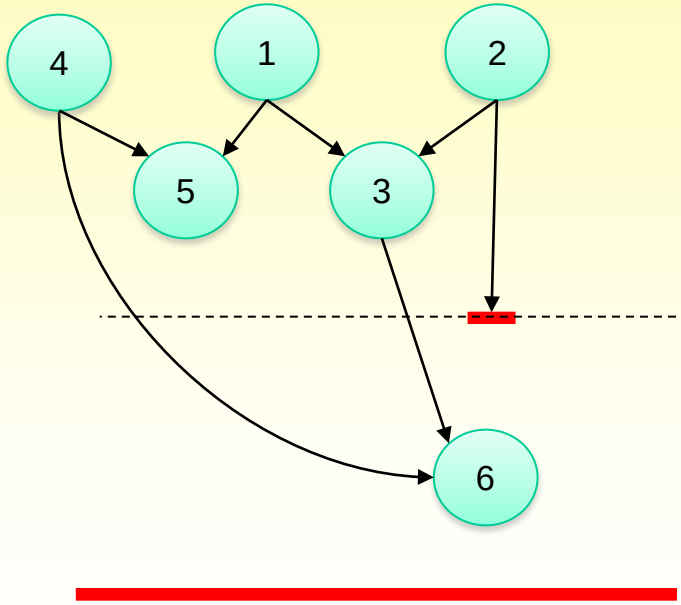


Partial control flow synchronization

#pragma taskwait on (expression)

- Expressions allowed are the same as for the directionality clauses
- Blocks 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];

main() {
  #pragma omp task input(A, B) inout(C)
  dgemm(A,B,C);    //1
  #pragma omp task input(D, E) inout(F)
  dgemm(D,E,F);    //2
  #pragma omp task input(C, F) inout(G)
  dgemm(C,F,G);    //3
  #pragma omp task input(A, D) inout(H)
  dgemm(A,D,H);    //4
  #pragma omp task input(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 input(C, H) inout(I)
  dgemm(H,G,C);    //6

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



Array sections

- Indicating as input/output/inout subregions of a larger structure:

  input (A[i])

  → the input argument is element *i* of A

- Indicating an array section:

  input ([BS]A)

  → the input argument is a block of size BS from address A

  input (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)

  input (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]

Array sections

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

=

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

=

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

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

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

=

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


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Array sections

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

=

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

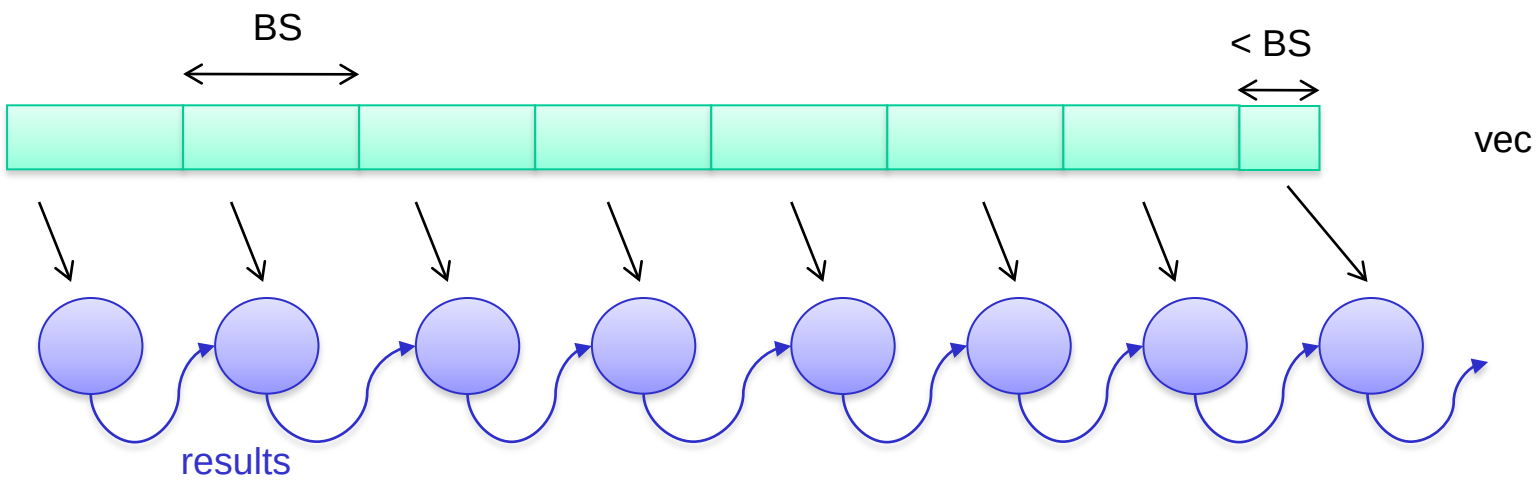
```
int a[N][M];
#pragma omp task input(a[2:3][3:4])
// 2 x 2 subblock of a at a[2][3]
```

=

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

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

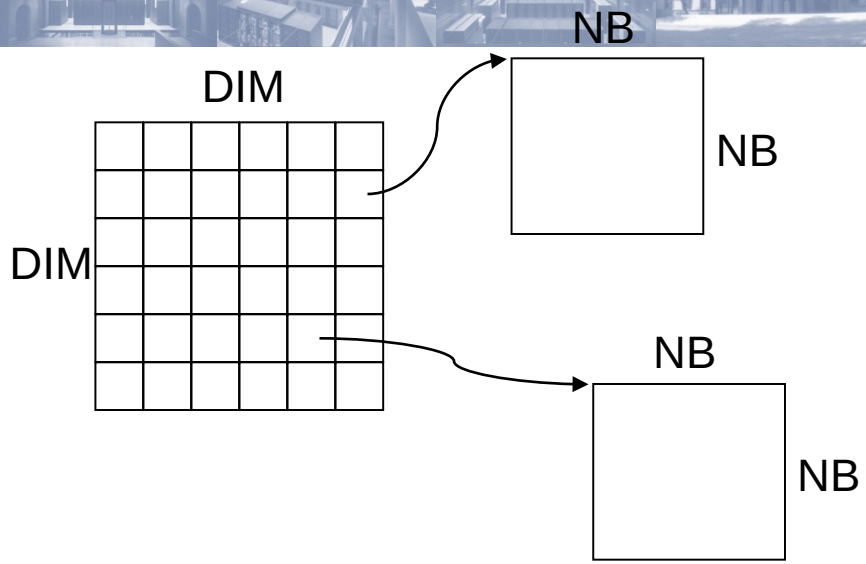
Array sections




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

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

   #pragma omp task input (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

Array sections




```
void matmul(double *A, double *B, double *C,
unsigned long NB)
{
  int i, j, k;

  for (i = 0; i < NB; i++)
            for (j = 0; j < NB; j++)
                      for (k = 0; k < NB; k++)

            C[i][j];+=A[i*NB+k]*B[k*NB+j];
}
```

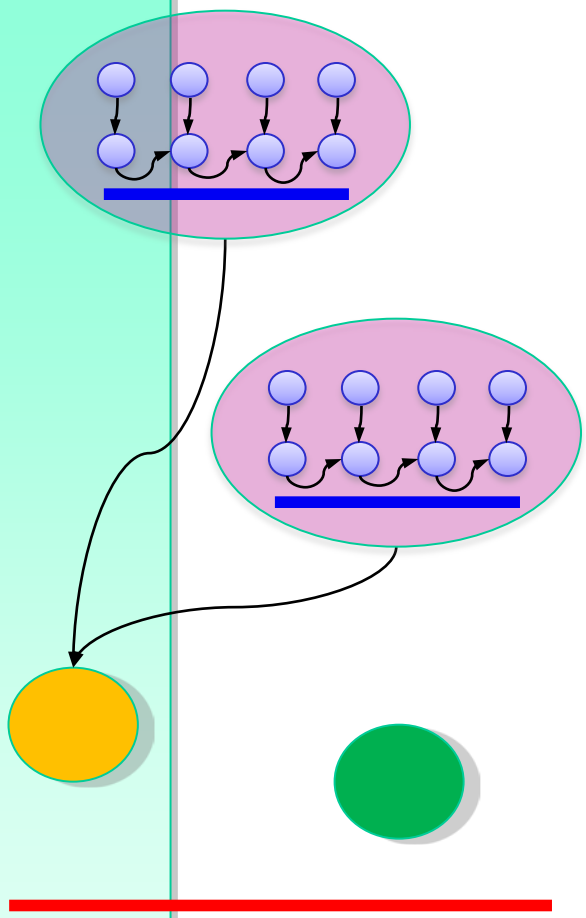
```
void compute(unsigned long NB, unsigned long DIM,
double *A[DIM][DIM], double *B[DIM][DIM], double *C[DIM][DIM])
{
  unsigned i, j, k;

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

Hierarchical task graph

- Nesting
  - Tasks can generate tasks themselves
- Hierarchical task dependences
  - Dependences only checked between siblings
  - Several task graphs (Hierarchy)
  - Effectively parallelizes task creation and dependence handling
- Different level tasks share the same resources
  - When ready, queued in the same queues
  - Currently, no priority differences between tasks and its children

Nesting

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

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

   for (int i=0; i<2; i++) {
      #pragma omp task output(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 input (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
}
```



Hierarchical task graph

- Nesting supports
  - Top down hierarchical parallelization
    - Every level contributes its potential parallelism
  - Modularity
    - Encapsulation
  - Effectively parallelizes task creation and dependence handling

Incomplete directionalities specification

- Directionality not required for all arguments
- May even be used with variables not accesed 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 input(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.

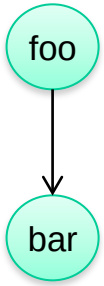
Will build proper dependences between tasks.

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

Incomplete directionalities specification

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

```
main () {
   int sentinel;

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

#pragma omp task input (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 copyin/out clauses cannot properly specify the address space that should be accessible in the devices

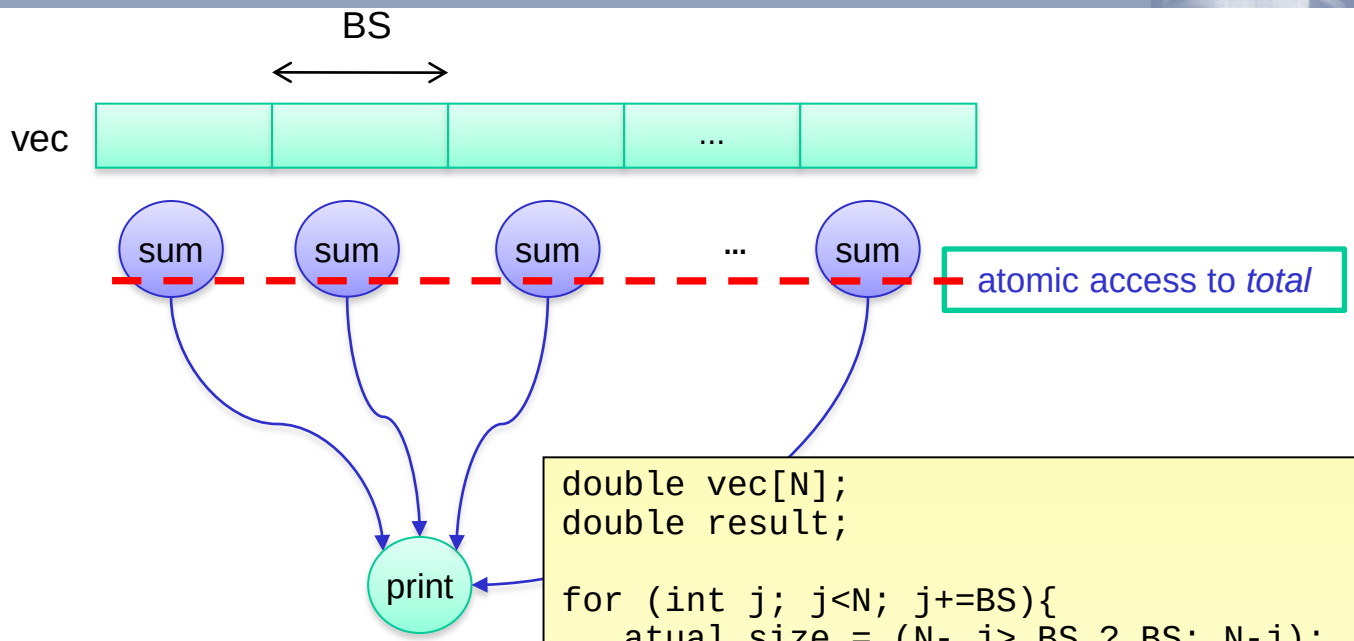
Concurrent

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

- 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.
    - Dependendences resulting from the regular input, output 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 concurrent chain using if needed OpenMP pragmas

```
#pragma omp atomic

#pragma omp critical
```

Concurrent




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

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

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

Heterogeneity

- Possible to specify that the code provided for a task is specific to a device (i.e. CUDA) and has to be run on the appropriate type of device (i.e. GPU)

- The device can have its own private physical address space
  - OmpSs copy clauses used to specify data accessed by the task whose consistency has to be maintained by the runtime

→ Section 3 in agenda

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**

Outlined tasks instantiation

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

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

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

int main()
{
   int X[100];

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

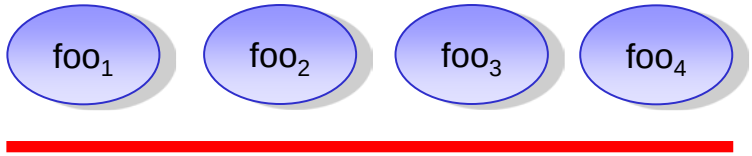
Invocation instantiates task

foo

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outlined tasks data scoping

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




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

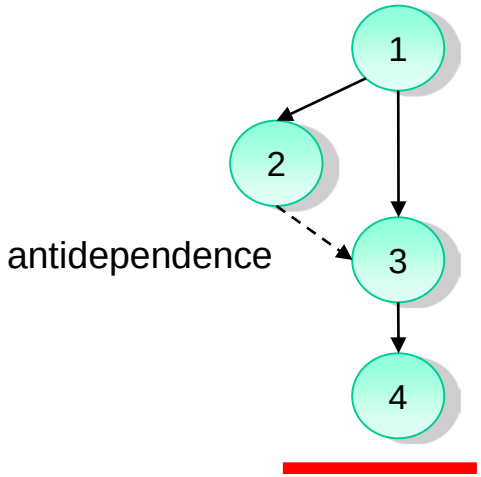
```
int Y[6]={1,2,3,4,5,6};

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

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

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

Defining dependences for outlined tasks

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

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

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

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

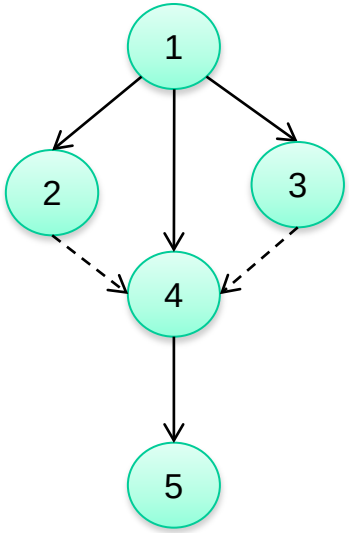
```
set(&x,5);       //1
do_print(&x);    //2
incr(&x);        //3
do_print(&x);    //4
#pragma omp taskwait
```



Mixing inlined and outlined tasks

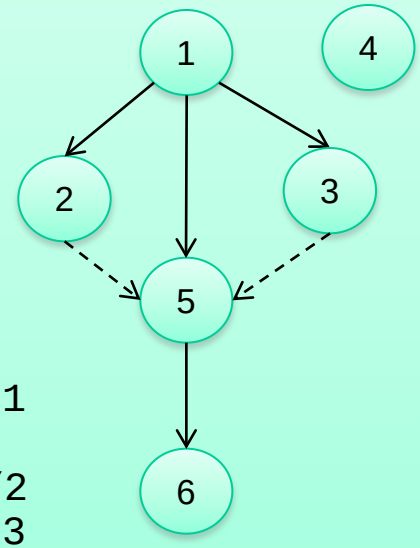
non-taskified: executed sequentially

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



Mixing inlined and outlined tasks

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

#pragma omp task input (x) // compiler warning, input clause discarded
void print_do (int x) {    // value captured at instantiation time
   printf(”from print_do %d\n" , x ) ;
}


int main()
{
int x;

x=3;
#pragma omp task output( x )
x = 5;                          //1
#pragma omp task input( x )
printf(”from main %d\n" , x );//2
do_print(&x);                   //3
print_do(x);                    //4
#pragma omp task inout( x )
x++;                            //5
#pragma omp task input( x )
printf (”from main %d\n" , x );//6
}
```



Output:

(4) from print_do 3

(2) from main 5

(3) from do_print 5

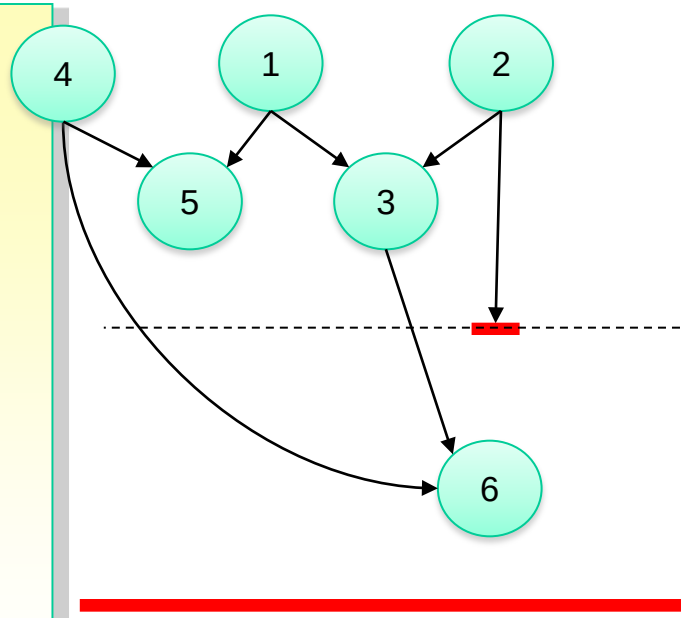
(6) from main 6

Partial control flow synchronization

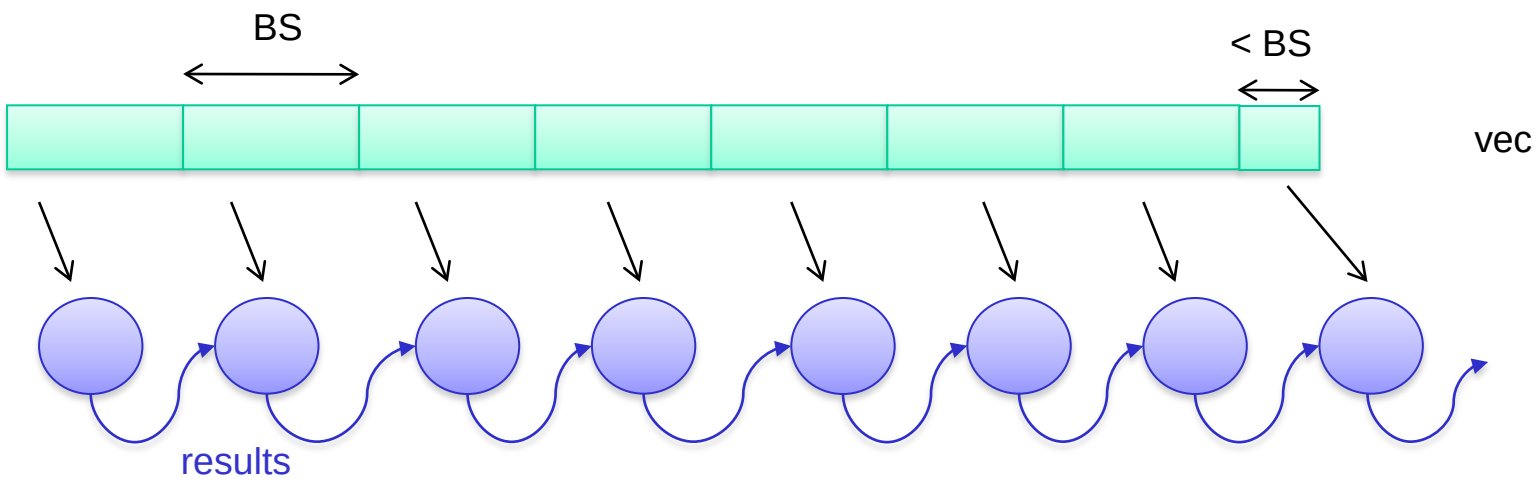
#pragma taskwait on (expression)

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

```
#pragma omp task input([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]);
}
```



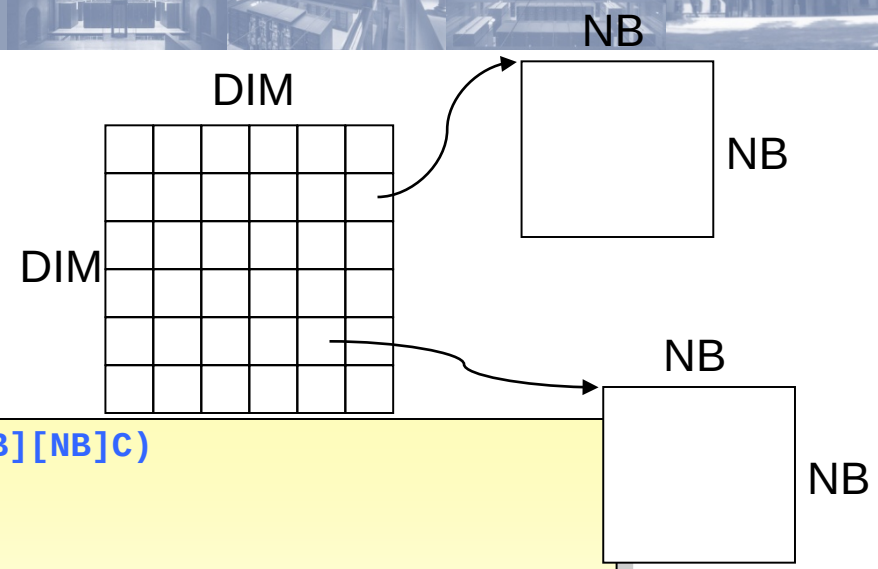
Array sections in outlined tasks




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

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, &total);
   }
}
```

dynamic size of argument

Array sections in outlined tasks

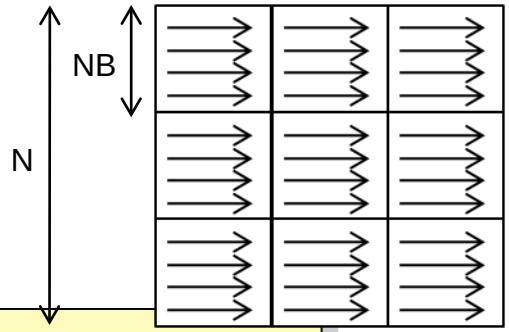



```
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void matmul(double *A, double *B, double *C,
unsigned long NB)
{
  int i, j, k;

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

```
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++)
        matmul (A[i][k], B[k][j], C[i][j], NB);
}
```

Array sections in outlined tasks

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



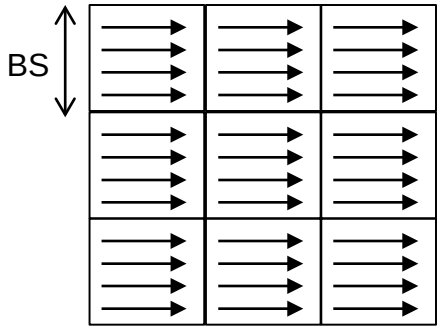

```
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void matmul(double  *A, double *B, double *C, unsigned long NB)
{
  int i, j, k;

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

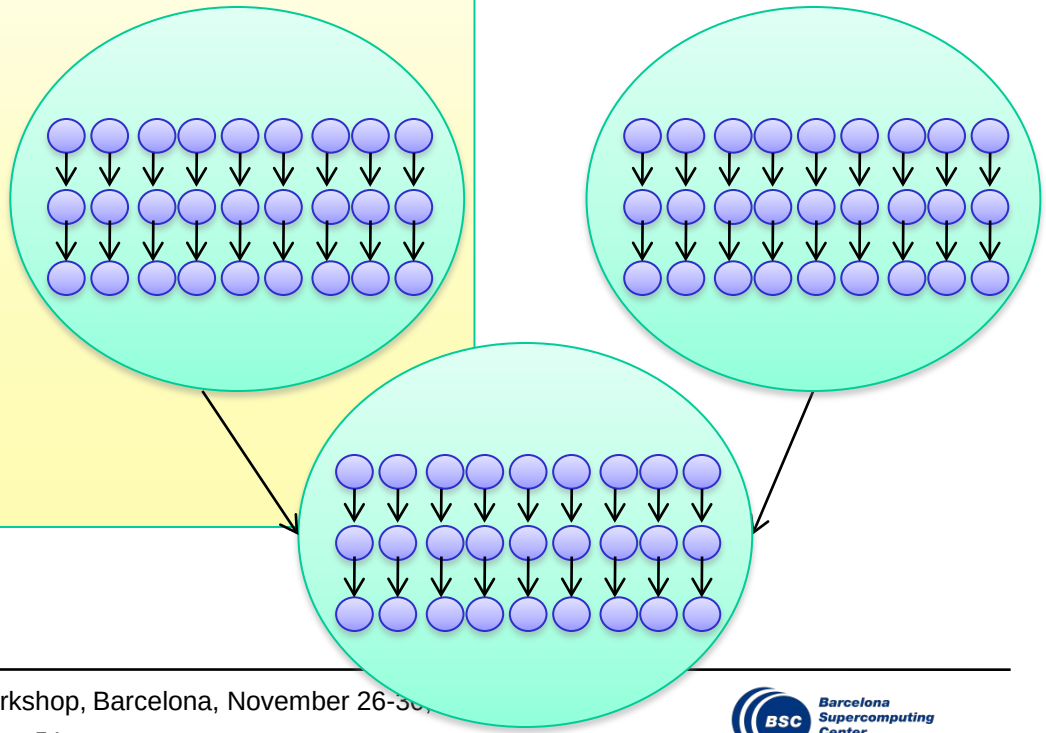
```
void compute(unsigned long NB, unsigned long DIM, double *A, double *B, double *C)
{
  unsigned i, j, k;

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

Hierarchical task graph

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

#pragma omp task input([N]A, [N]B) output([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_dgem(&A[i][k*BS], &B[k][j*BS], &C[i][j*BS]);
#pragma omp taskwait

}

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

}
```

Block data-layout





Incomplete directionalities specification

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

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

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

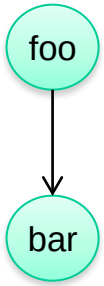
Using element C[0][0] as representative/sentinel for the whole block.

Will build proper dependences between tasks.

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

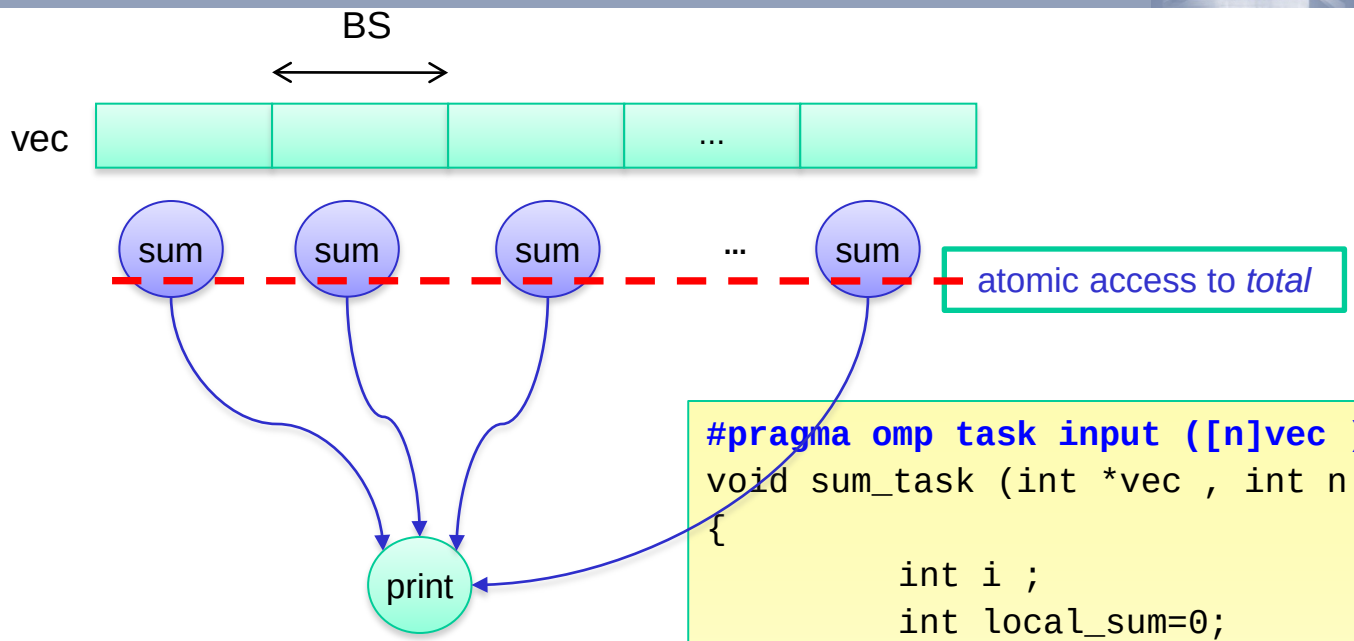
Incomplete directionalities specification

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

#pragma omp task input (*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

Concurrent in outlined tasks

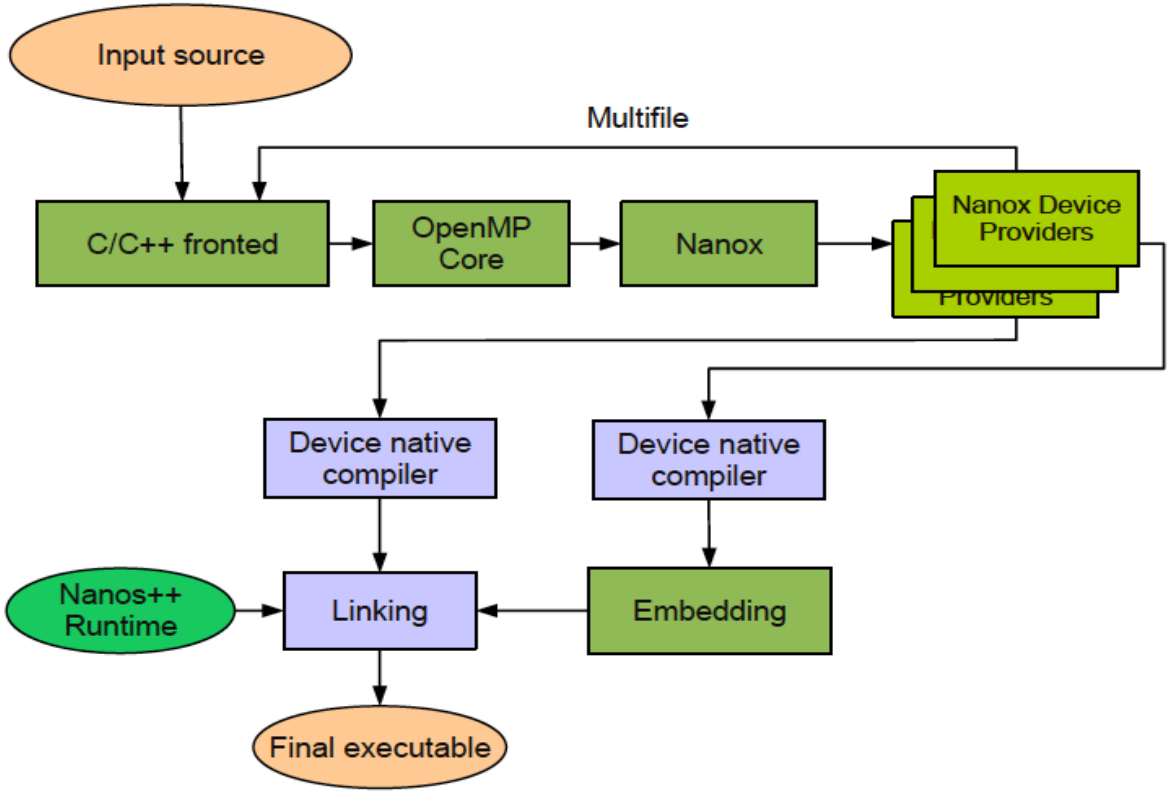



```
#pragma omp task input ([n]vec ) concurrent (*results)
void sum_task (int *vec , int n , int  *results)
{
          int i ;
          int local_sum=0;

          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 input (total)
printf (“TOTAL is %d\n”, total);
}
```

Infrastructure

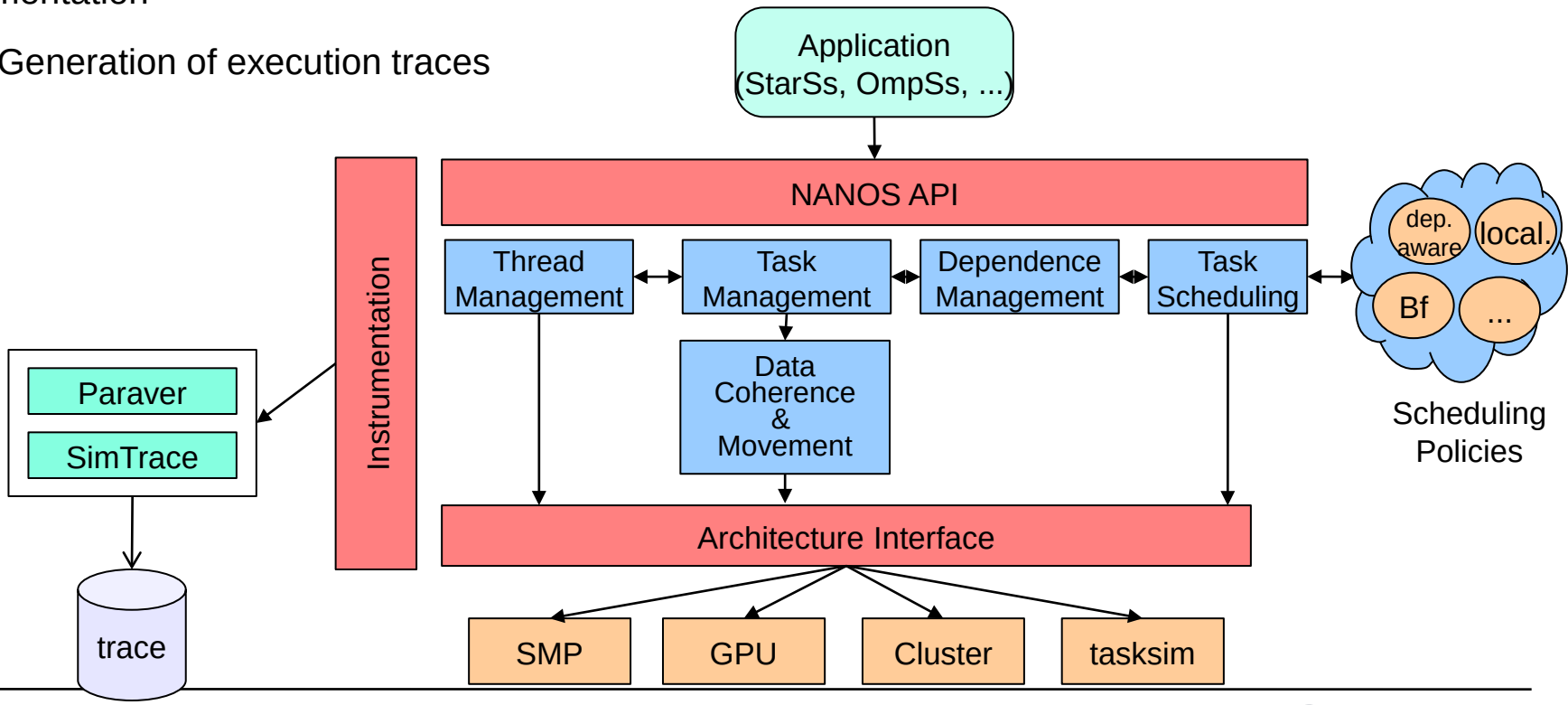
The Mercurium Compiler

- 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



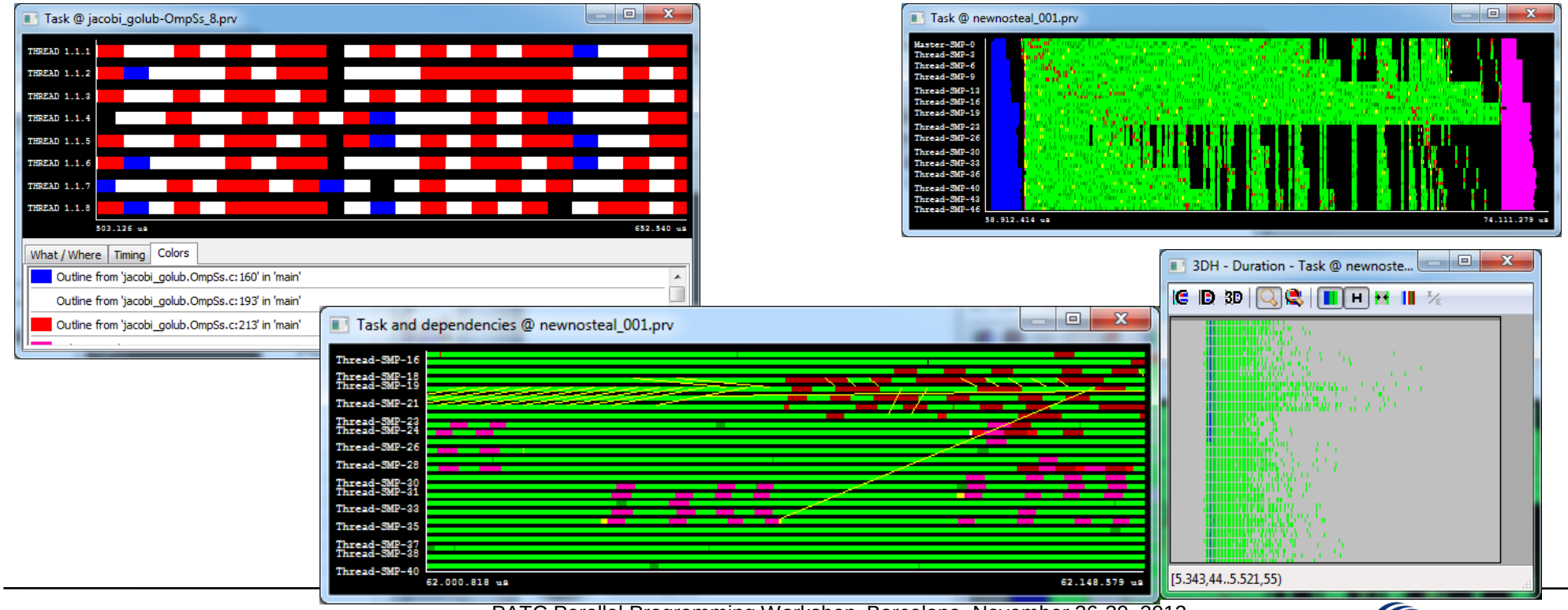
The NANOS++ Runtime

- Support to different programming models: OpenMP (OmpSs), StarSs, Chapel
- Independent components for thread, task, dependence management, task scheduling, ...
- Most of the runtime independent of the target architecture: SMP, GPU, tasksim simulator, cluster
- Support to heterogeneous targets
  - → i.e., threads running tasks in regular cores and in GPUs
- Instrumentation
  - →Generation of execution traces



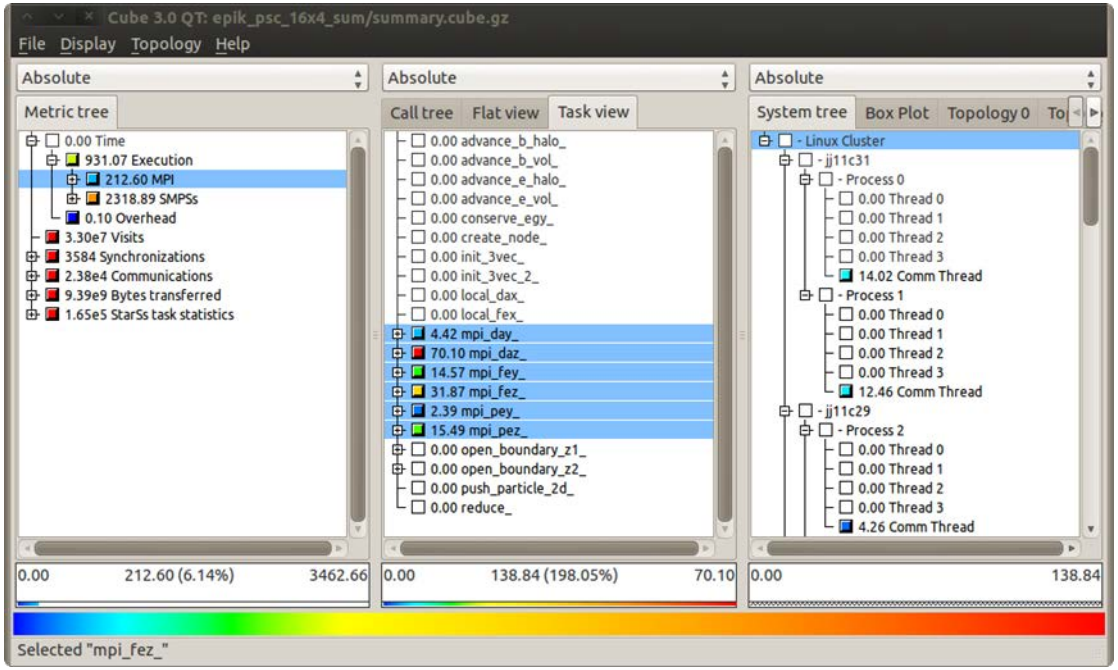
Performance tools: Paraver

- Probe hooks provided by compiler and runtime
- Integrated into the general Extrae MPI instrumentation environment
- Very detailed statistics and analysis of time behavior



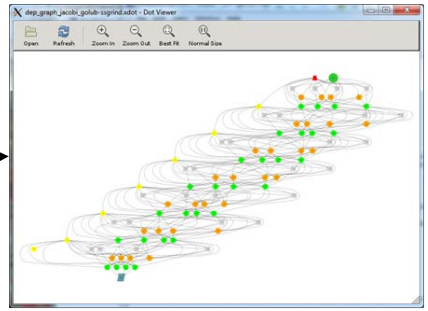
Performance tools: Scalasca

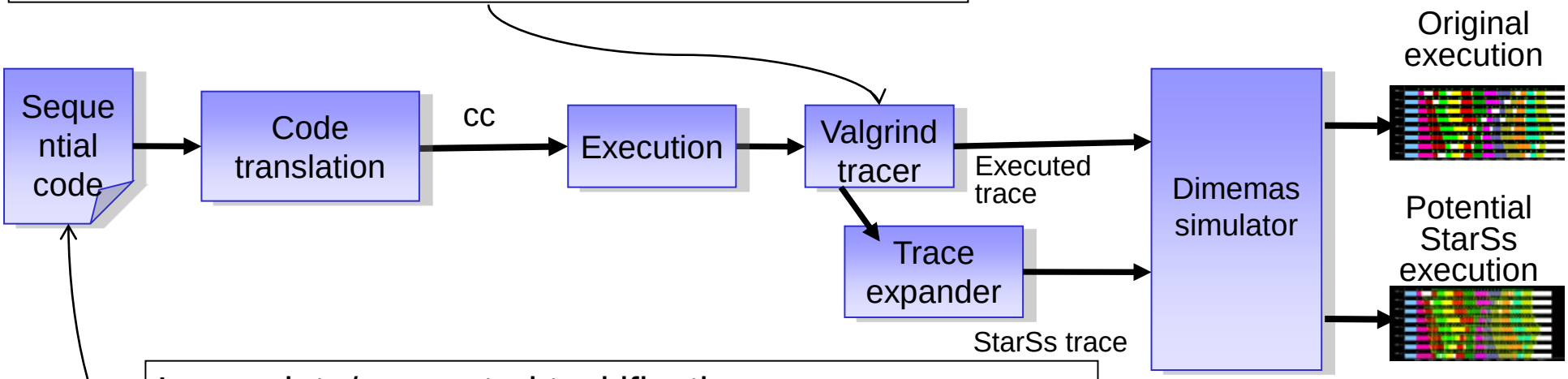
- Probe hooks provided by compiler and runtime
- Inject scalasca probes
- Specific task oriented statistics



Predictive tools: Tareador

Valgrind based instrumentation
- Track task entry and exit
- Track mallocs
- Track ALL memory accesses → dependences
- Estimate time = f(#instructions)



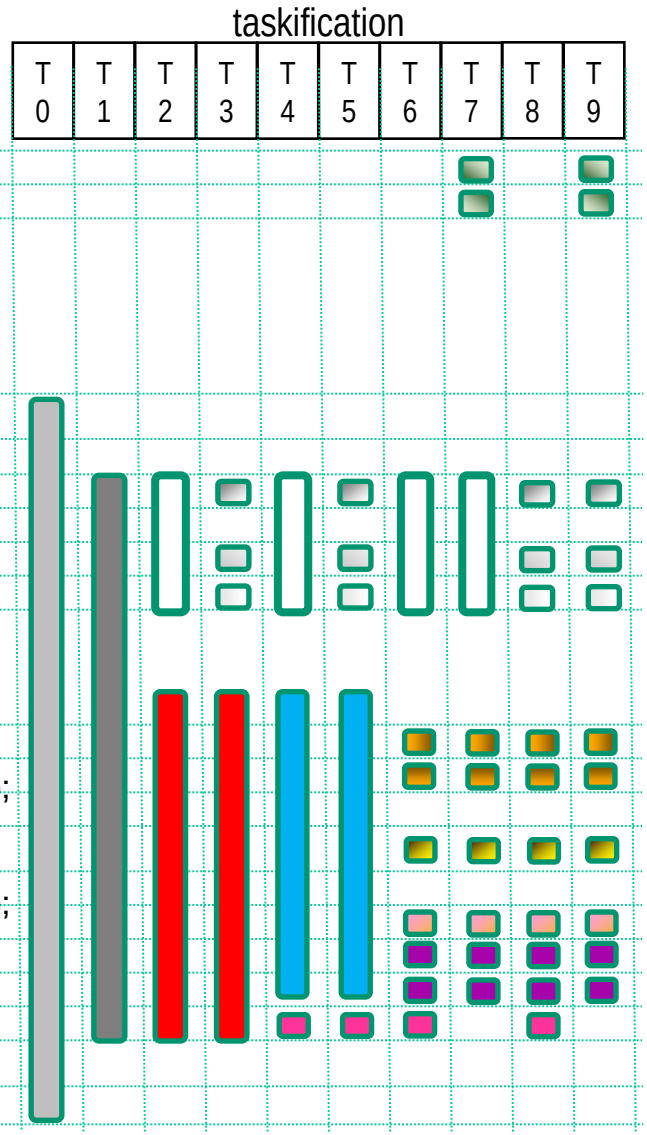


Incomplete/suggested taskification
- Compiler directives
- Runtime calls (even on not well structured code)

"Quantifying the potential task-based dataflow parallelism in MPI applications". V. Subotic et all. Europar 2011

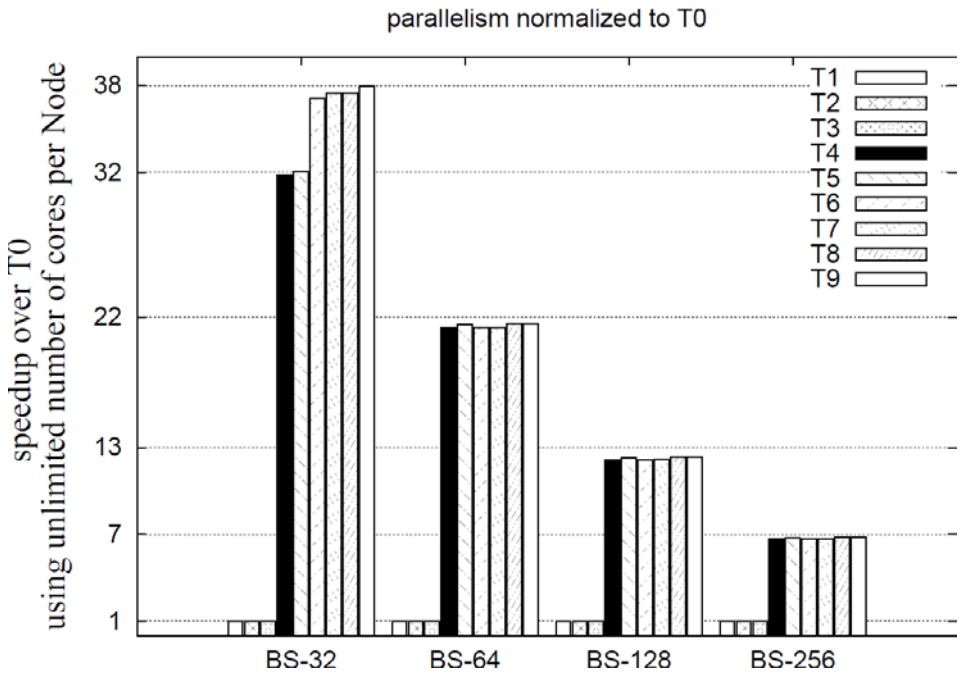
Predictive tools: Tareador

```
void update( ... ) {
  HPL_dtrsm( ... );
  HPL_dgemm(... );
}

main() {

...
  for( j = 0; j < N; j += BS )
  {
    panel_init( ... );
    if (cond0)
      fact( ... );
    init_for_pivoting( ... );
    for(i = k; i < P; i+= BS) {

      if(cond1)
        HPL_dlaswp01N( ... );
      HPL_spreadN( ... );
      if(cond2)
        HPL_dlaswp06N( ... );
      if(cond3)
        HPL_dlacpy( ... );
      HPL_rollN( ... );
      HPL_dlaswp00N( ... );
      update( ... );
    }
  }
...
}
```



*HP Linpack:
4 MPI processes;
Problem size: 8192;
Block size: 256,128, 64, 32.

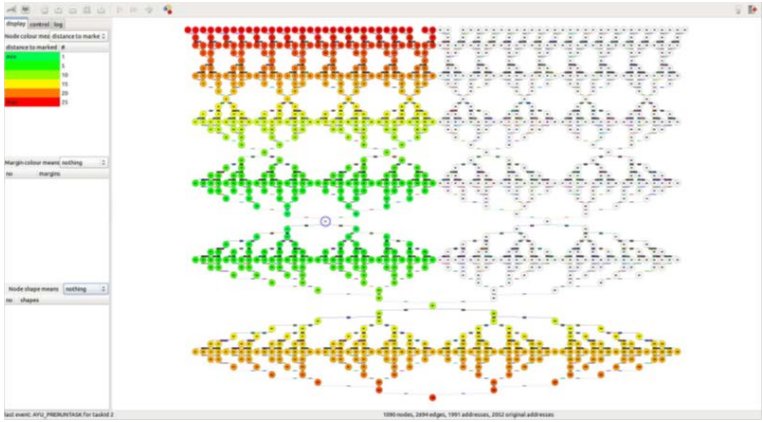
The parallelism is released in transition T3 -> T4 when function *update* is separated from the rest of the inner loop



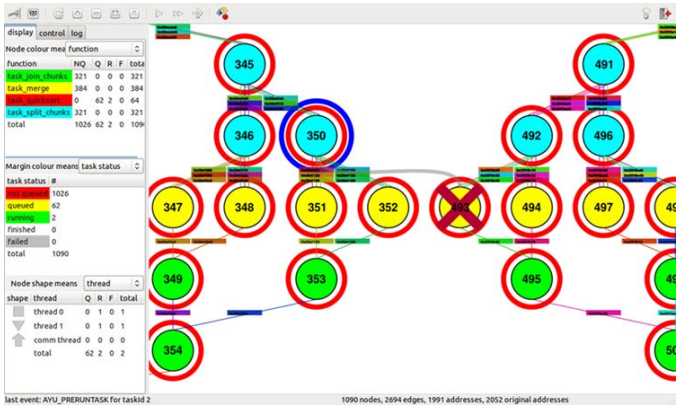
Debugging: AYUDAME/TEMANEJO

- Leverage probe hooks provided by compiler and runtime

- Task based debugging:
  - Display graph
  - Control execution environment (#threads,…)
  - Breakpoints at tasks



- Interface to instruction level debugger (gdb)



Conclusion

OmpSs: Enabler for exascale

- Can exploit very unstructured parallelism
  - Not just loop/data parallelism
  - Easy to change structure
- Supports large amounts of lookahead
  - Not stalling for dependence satisfaction
- Allow for locality optimizations to tolerate latency
  - Overlap data transfers, prefetch
  - Reuse
- Nicely hybridizes into MPI/StarSs
  - Propagates to large scale the node level dataflow characteristics
  - Overlap communication and computation
  - A chance against Amdahl's law

- Support for heterogeneity
  - Any # and combination of CPUs, GPUs
  - Including autotuning
- Malleability: Decouple program from resources
  - Allowing dynamic resource allocation and load balance
  - Tolerate noise

Data-flow; Asynchrony

Potential is there;
Can blame runtime

Compatible with proprietary low level technologies