Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

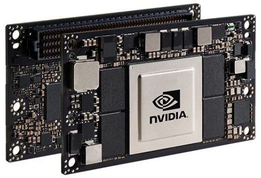
Static analysis to enhance programmability and performance in OmpSs-2

Adrian Munera, Sara Royuela, Eduardo quiñones

C3PO'20: Compiler-assisted Correctness Checking and Performance Optimization for HPC
Held in conjunction with **ISC 2020**

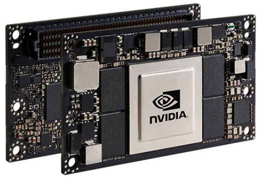
The need for Parallel Programming Models

Hardware Heterogeneity

HPC	 NVIDIA TitanV (5120 CUDA cores)	 Intel Xeon Phi KNL (72-core fabric)
Embedded systems	 NVIDIA Jetson (512 CUDA cores)	 Kalray MPPA (256-core fabric)

The need for Parallel Programming Models

Hardware Heterogeneity

HPC	 NVIDIA TitanV (5120 CUDA cores)	 Intel Xeon Phi KNL (72-core fabric)
Embedded systems	 NVIDIA Jetson (512 CUDA cores)	 Kalray MPPA (256-core fabric)

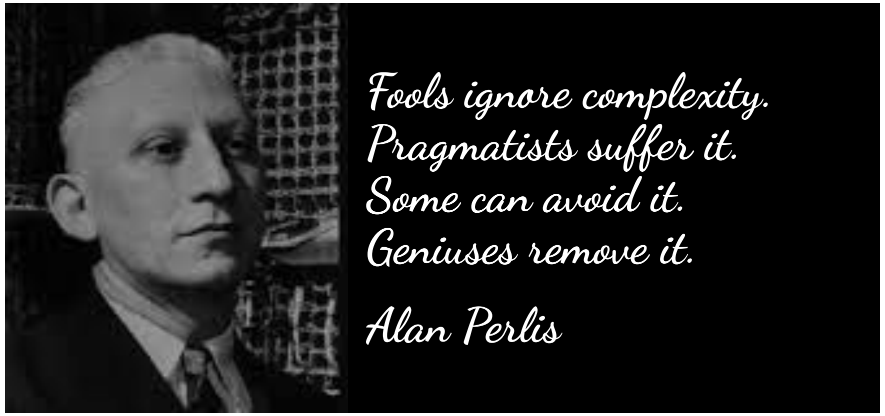
Parallel Programming Models productivity

Programmability	Abstracts the parallelism while hiding the complexities of the underlying computing platform
Portability	The same source code is valid in different platforms, including SMP and heterogeneous systems
Performance / Scalability	Rely on run-time mechanisms to exploit the performance capabilities of parallel platforms

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

The programming models melting pot

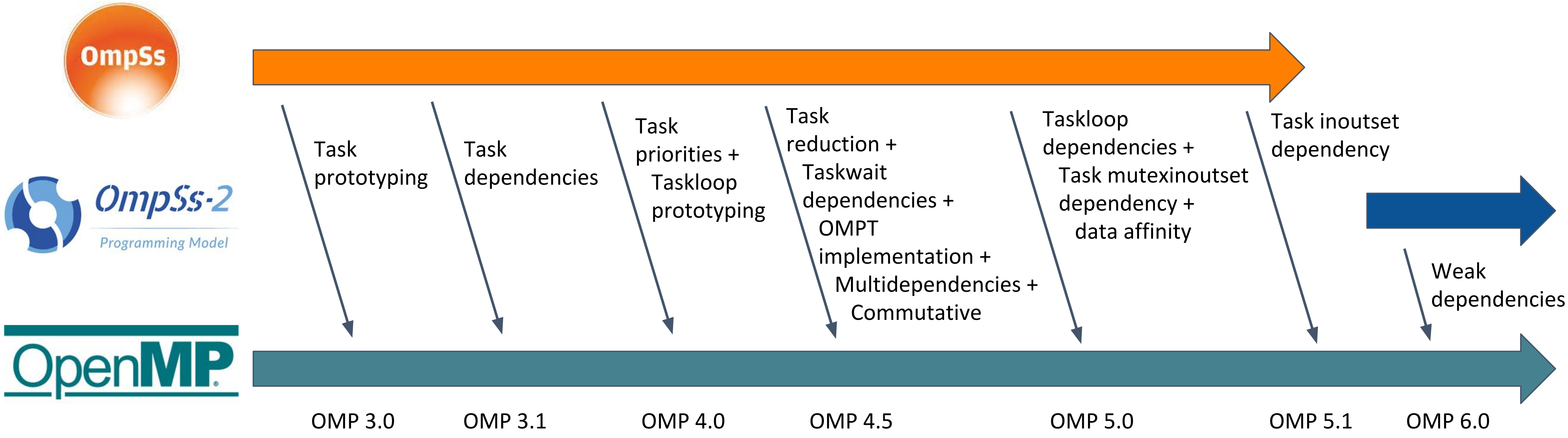
- Several programming models targeting productivity coexist: *NVIDIA CUDA, Intel TBB, OpenCL, Cilk++, C++11, OpenACC, OpenMP, OmpSs, Pthreads, etc*.
- Productivity is recipe for success:
  - **High-level APIs** are less complex and entail mild learning curves.
  - Models based on **compile-time directives** allow incremental parallelization, without sacrificing portability and programmability.
  - **Task-based models** offer the flexibility needed for dynamic and unstructured applications.



OmpSs-2 and the future OpenMP

Introducing changes in the OpenMP specification is a long-distance race.

The main goal of OmpSs/OmpSs-2 is **fast-prototyping tasking features** to include them in OpenMP.



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outline

- **Introduction to OmpSs-2**
- Proposed algorithms for programmability and performance
  - Auto-scope + Evaluation
  - Auto-release + Evaluation
- Implementation
- Discussion

OmpSs-2: execution model

- OmpSs-2 is a ***thread-pool*** based model: parallelism is spawned when the application starts and joined when it finishes.

```
int main (void) {
// Thread pool spawns, like #pragma omp parallel
// Only one thread executes, like #pragma omp single
int x[10], a;
for (int i = 0; i<4; ++i) {
}

return 1;
// Thread pool is joined, like #pragma omp barrier
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs-2: execution model

- OmpSs-2 is a ***thread-pool*** based model: parallelism is spawned when the application starts and joined when it finishes.
- When a thread of the program encounters a `task` construct, it creates a ***task***. That can be executed by any of the spawned threads.

```
int main (void) {
// Thread pool spawns, like #pragma omp parallel
// Only one thread executes, like #pragma omp single
int x[10], a;
for (int i = 0; i<4; ++i) {
  // Creation of tasks
  #pragma oss task
  {...}
  #pragma oss task
  {...}
}

return 1;
// Thread pool is joined, like #pragma omp barrier
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs-2: execution model

- OmpSs-2 is a ***thread-pool*** based model: parallelism is spawned when the application starts and joined when it finishes.
- When a thread of the program encounters a `task` construct, it creates a ***task***. That can be executed by any of the spawned threads.
- Tasks include **data-sharing attributes** to define the view of the memory for each variable.

```
int main (void) {
// Thread pool spawns, like #pragma omp parallel
// Only one thread executes, like #pragma omp single
int x[10], a;
for (int i = 0; i<4; ++i) {
  // Creation of tasks
  #pragma oss task shared(x)
  {...}
  #pragma oss task shared(x)
  {...}
}

return 1;
// Thread pool is joined, like #pragma omp barrier
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs-2: execution model

- OmpSs-2 is a ***thread-pool*** based model: parallelism is spawned when the application starts and joined when it finishes.

- When a thread of the program encounters a `task` construct, it creates a ***task***. That can be executed by any of the spawned threads.

- Tasks include **data-sharing attributes** to define the view of the memory for each variable.

- Tasks can be synchronized by means of task **dependency clauses** and **synchronization constructs** (e.g., taskwait).

```
int main (void) {
// Thread pool spawns, like #pragma omp parallel
// Only one thread executes, like #pragma omp single
int x[10], a;
for (int i = 0; i<4; ++i) {
  // Creation of tasks
  #pragma oss task shared(x) out(x[i])
  {...}
  #pragma oss task shared(x) in(x[i])
  {...}
}
#pragma oss taskwait
return 1;
// Thread pool is joined, like #pragma omp barrier
}
```

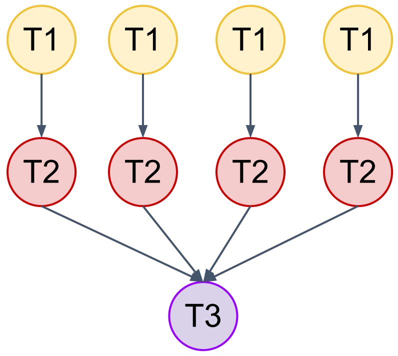
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs-2: execution model

- OmpSs-2 is a ***thread-pool*** based model: parallelism is spawned when the application starts and joined when it finishes.
- When a thread of the program encounters a `task` construct, it creates a ***task***. That can be executed by any of the spawned threads.
- Tasks include **data-sharing attributes** to define the view of the memory for each variable.
- Tasks can be synchronized by means of task **dependency clauses** and **synchronization constructs** (e.g., taskwait).

```
int main (void) {
// Thread pool spawns, like #pragma omp parallel
// Only one thread executes, like #pragma omp single
int x[10], a;
for (int i = 0; i<4; ++i) {
  // Creation of tasks
  #pragma oss task shared(x) out(x[i]) label(T1)
  {...}
  #pragma oss task shared(x) in(x[i]) label(T2)
  {...}
}
#pragma oss taskwait label(T3)
return 1;
// Thread pool is joined, like #pragma omp barrier
}
```

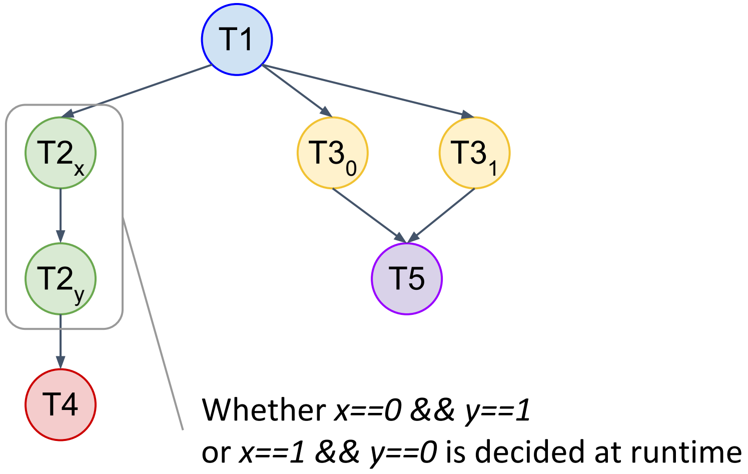
This generates a Task Dependency Graph



OmpSs-2: task dependencies

- **`in`**, **`out`**, and **`inout`** clauses (same as in OpenMP)
- **`concurrent`:** This clause is like inout, but allows parallelism across tasks with a concurrent dependency of the same object (extra synchronizations, like atomics, might be needed).
- **`commutative`:** This clause also acts as inout, but allows any ordering between tasks with the same commutative dependency.
- **Taskwait dependencies:** The taskwait construct accepts dependencies, and acts as an empty task.

```
#pragma oss task out(a) out(b) label(T1)
{...}
for (int i = 0; i<2; ++i)
  #pragma oss task commutative(a) label(T2)
  {...}
for (int i = 0; i<2; ++i)
  #pragma oss task concurrent(b) label(T3)
  {...}
#pragma oss taskwait in(a) label(T4)
#pragma oss taskwait in(b) label(T5)
```



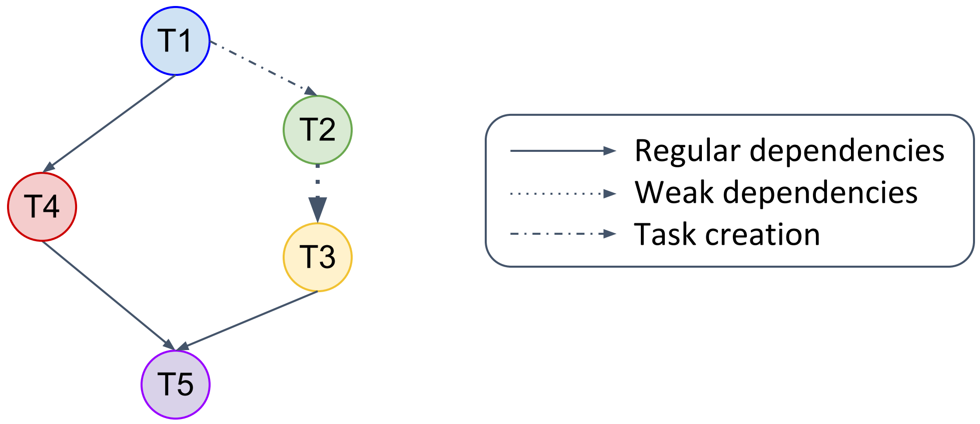
OmpSs-2: nested tasks and dependencies

- A task defines two types of dependencies:
  - data accessed by the task: *regular dependencies*.
  - data accessed by subtasks: ***weak dependencies***.
- **Linked dependency domains** between parent and children tasks:

How
- *regular dependencies* are released when the task finishes
- *weak dependencies* are released when the corresponding children task has finished

Why
- avoid *data races* between tasks with different parents.

```
int x, y;
#pragma oss task weakout(x) out(y) label(T1)
{
  #pragma oss task out(x) label(T2)
  { x = 1; }
  y = 2;
}
#pragma oss task in(x) label(T3)
{ assert(x == 1); }
#pragma oss task in(y) label(T4)
{ assert(y == 2); }
#pragma oss taskwait label(T5)
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OmpSs-2: early release of dependencies

OmpSs-2 allows executing tasks to early release their dependencies using the **`release` directive**.

This provides fine-grained control for tasks that describe a large set of data-dependencies that are processed in chunks.

T2 consuming *data[i;chunk_size]* can start as soon as release *data[i;chunk_size]* is executed, although T1 has not finished yet

```
int data[size];

#pragma oss task out(data[0; size]) label(T1)
{
 for (int i = 0; i < size; i += chunk_size) {
   for (int j = i; j < chunk_size; j++)
       fast_process(&data[j]);
   }
   #pragma oss release out(data[i;chunk_size])
 }
 for (int i = 0; i < size; i += chunk_size) {
  #pragma oss task in(data[i; chunk_size]) label(T2)
  {
   slow_process(&data[i], chunk_size);

 }
}
```

Outline

- Introduction to OmpSs-2
- Proposed algorithms for programmability and performance
  - **Auto-scope + Evaluation**
  - Auto-release + Evaluation
- Implementation
- Discussion

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

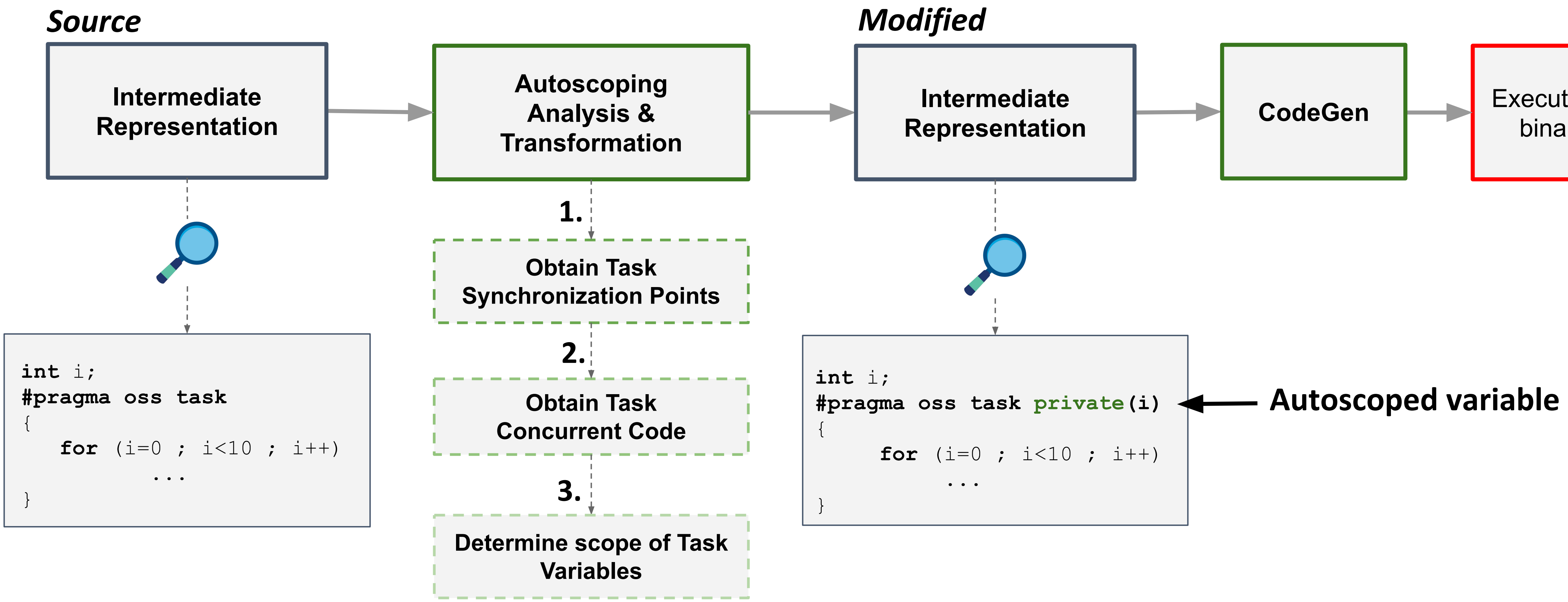
Auto-scope usefulness: related work

Manual scoping variables is *arduous* and *error-prone* → **Programmability**, **Correctness**

Several works tackle the automatic scope of variables in OpenMP, among others:

1. Lin et. al, *Automatic scoping of variables in parallel regions of an OpenMP program*, IWOMP 2004
   - a set of rules to accomplish auto-scope in parallel regions
   - obtained the same performance as user directives.
2. Royuela et. al, *Auto-scoping for OpenMP tasks*, IWOMP 2012
   - a set of rules to accomplish auto-scope in tasks obtained a 85% success
   - compared to same feature in Oracle Solaris Studio 12.3, with a 78% success
3. Wang et.al, *Automatic scoping of task clauses for the openmp tasking model*, Journal of Supercomputing, 2015
   - simpler set of rules using synchronizations between tasks
   - better success ratio, but poor performance

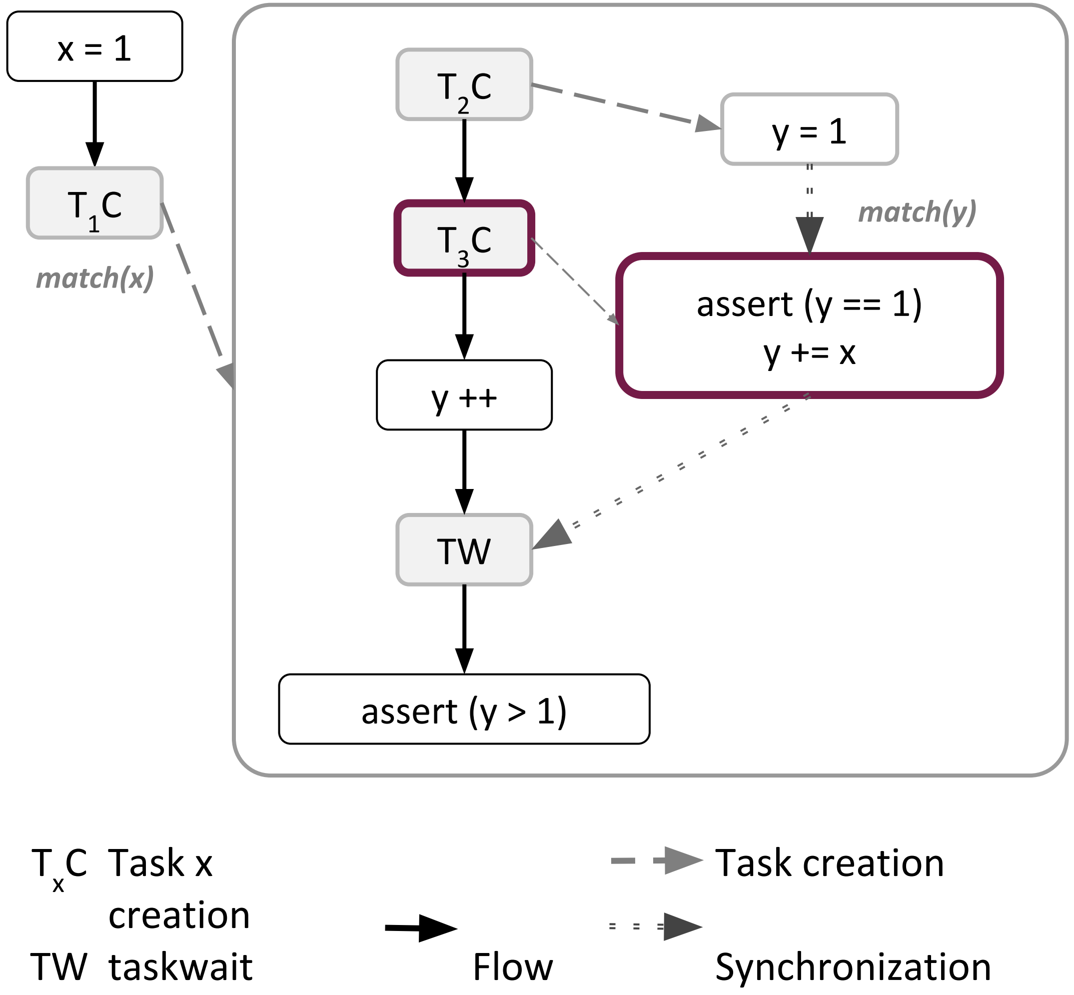
Auto-scope overview



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Auto-scope by example

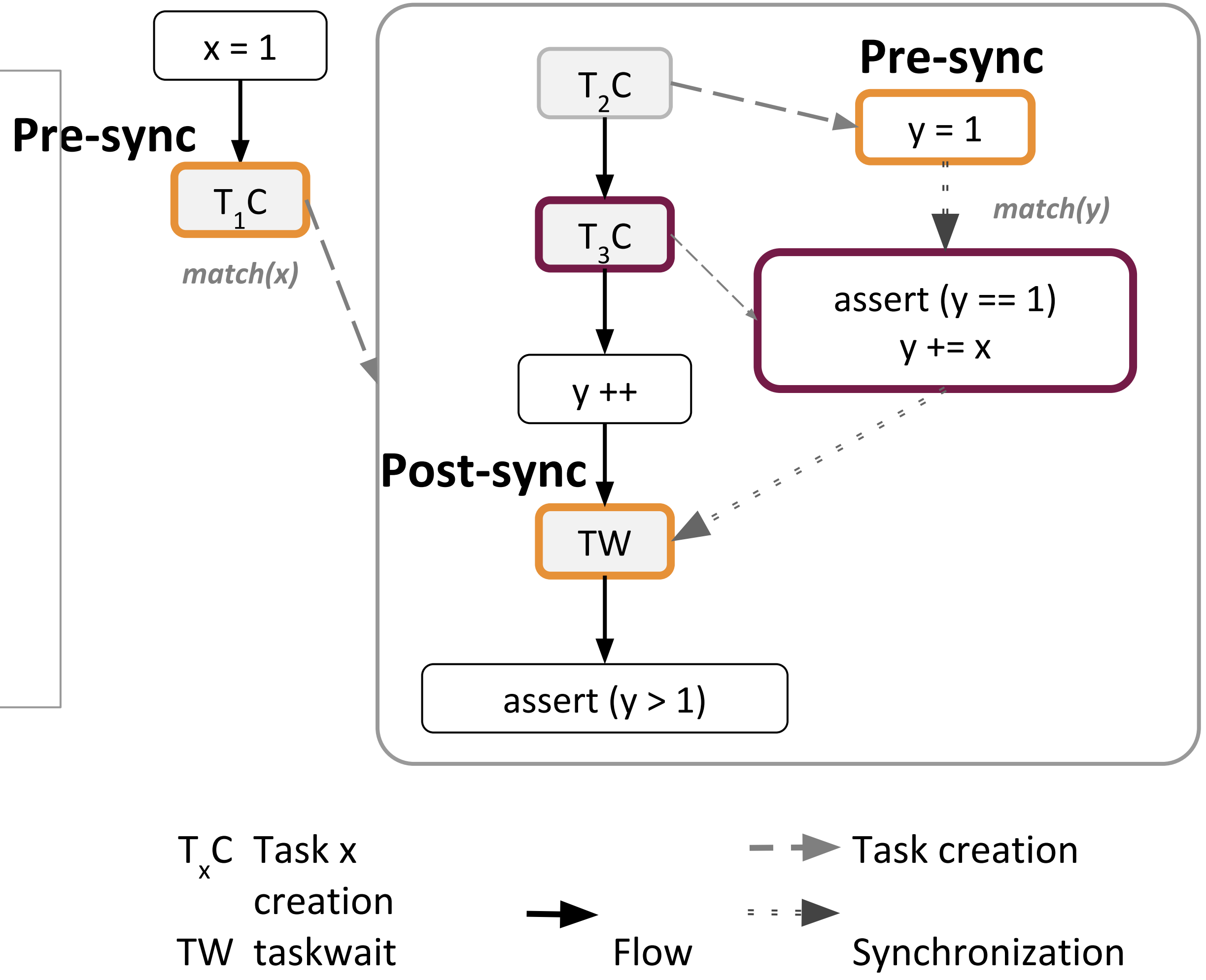
```
int x = 1;
#pragma oss task weakin(x) out(y) label(T1)
{
  #pragma oss task out(y) label(T2)
  { y = 1; }
  #pragma oss task inout(y) in(x) label(T3)
  {
    // Here, y might be 1 or 2
    assert(x == 1);
    y += x;
  }
  y ++;
  #pragma oss taskwait in(y)
  assert(y == 3);
}
```



Auto-scope by example

Step 1: Synchronization points

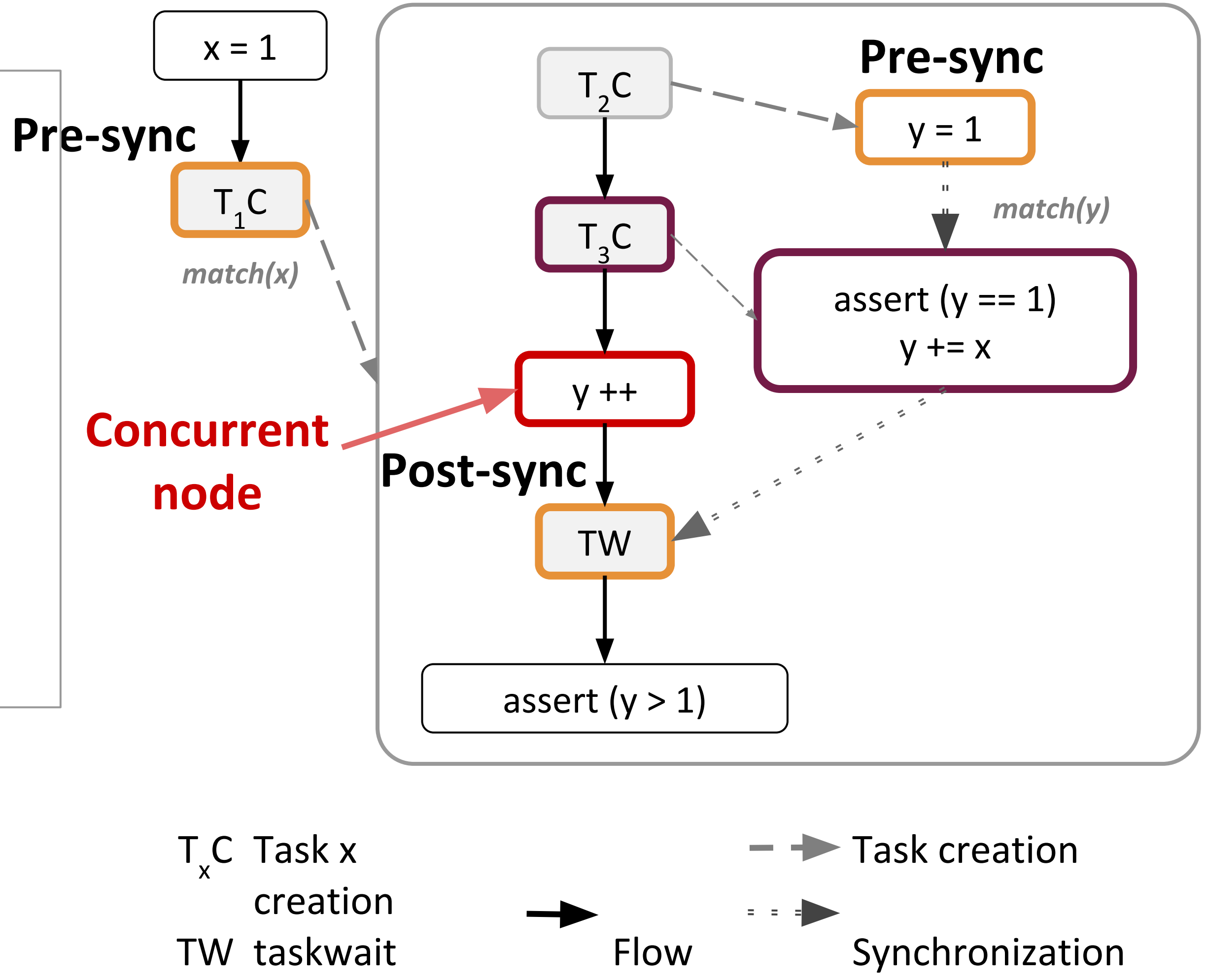
```
int x = 1;
#pragma oss task weakin(x) out(y) label(T1)
{
  #pragma oss task out(y) label(T2)
  { y = 1; }
  #pragma oss task inout(y) in(x) label(T3)
  {
    // Here, y might be 1 or 2
    assert(x == 1);
    y += x;
  }
  y ++;
  #pragma oss taskwait in(y)
  assert(y == 3);
}
```



Auto-scope by example

Step 2: Concurrent regions

```
int x = 1;
#pragma oss task weakin(x) out(y) label(T1)
{
  #pragma oss task out(y) label(T2)
  { y = 1; }
  #pragma oss task inout(y) in(x) label(T3)
  {
    // Here, y might be 1 or 2
    assert(x == 1);
    y += x;
  }
  y ++;
  #pragma oss taskwait in(y)
  assert(y == 3);
}
```

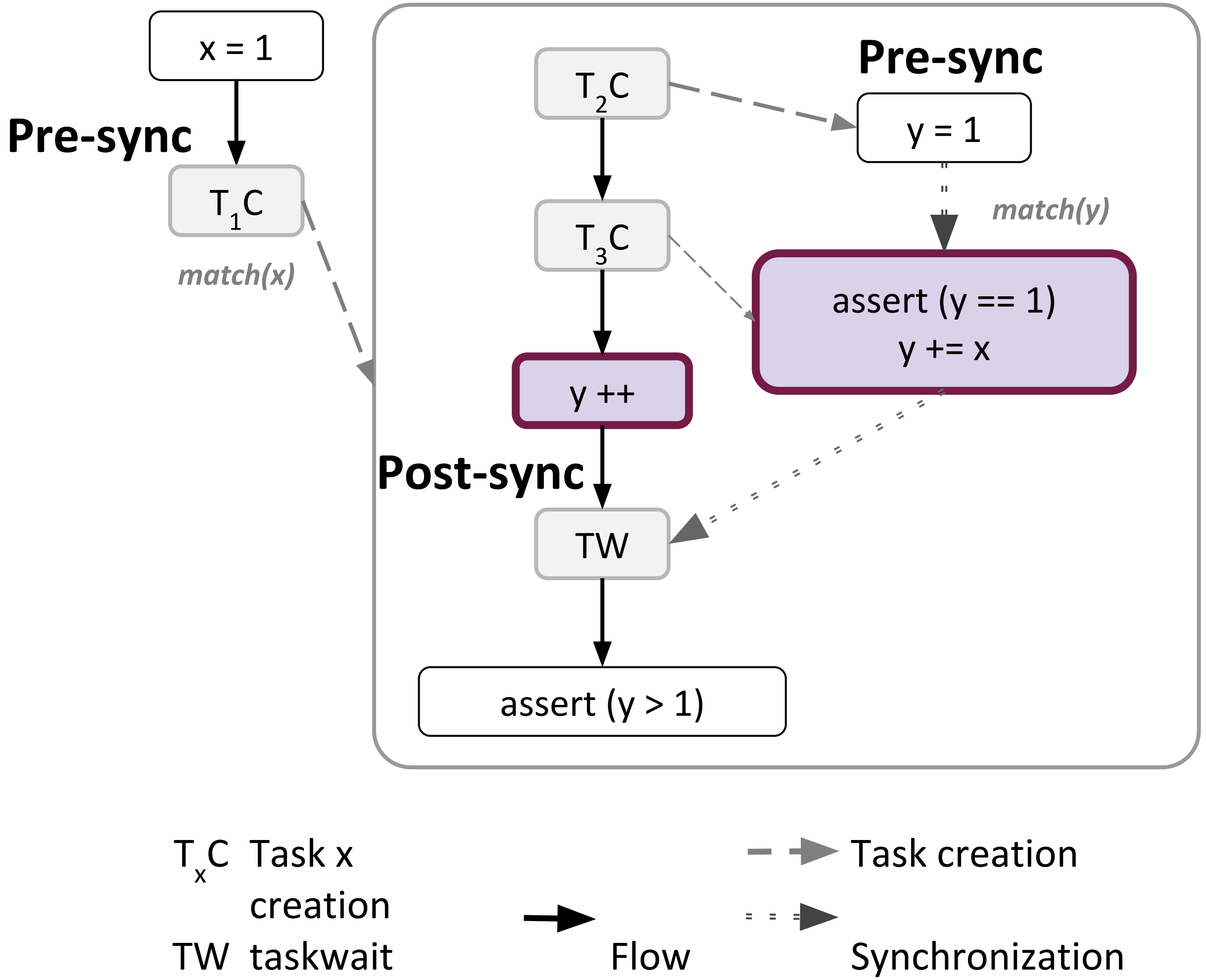


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Auto-scope by example

Step 3: Final data sharings

Variables' use				
Variable	**T3**		**Concurrent T3**	
	Read	Written	Read	Written
X	**Yes**	**No**	**No**	**No**
Y	**Yes**	**Yes**	**Yes**	**Yes**



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Auto-scope by example

Step 3: Final data sharings

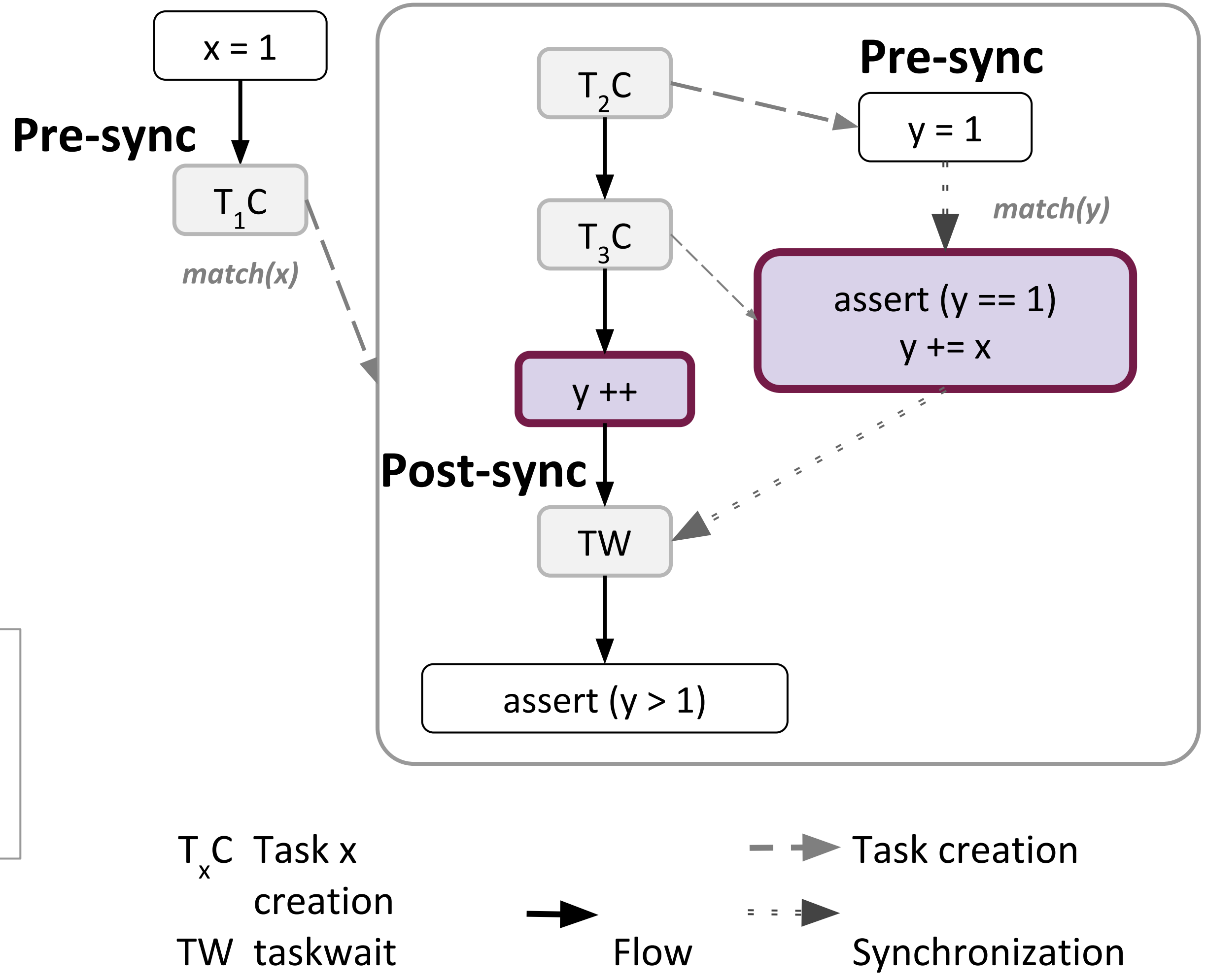
Variables' use				
Variable	**T3**		**Concurrent T3**	
	Read	Written	Read	Written
X	**Yes**	**No**	**No**	**No**
Y	**Yes**	**Yes**	**Yes**	**Yes**

Using the set of rules as defined in:
Royuela, S., Duran, A., Liao, C., & Quinlan, D. J., *Auto-scoping for OpenMP tasks*, IWOMP 2012

T3 data-sharing attributes:

x → **`firstprivate`**

y → *race condition* → **`firstprivate`**



Auto-scope results

Benchmarks	Description			LLVM Results				
	#tasks	nested tasks	method	shared	private	firstprivate	undefined	(%) success
Alignment	1	no	iter	2	4	14	0	100%
FFT	41	no	rec	102	0	140	0	100%
Fib	2	no	rec	2	0	3	0	100%
Health	2	yes	iter&rec	1	1	3	0	100%
Floorplan	1	no	iter&rec	3	1	9	2	86.66%
Nqueens	1	no	iter&rec	2	0	4	0	100%
Sort	9	yes	rec	27	0	10	0	100%
SparseLU	4	yes	iter	4	3	11	0	100%
UTS	2	no	iter&rec	2	1	3	0	100%
Cholesky	4	no	iter	4	0	12	0	100%
Saxpy	2	yes	iter	4	0	3	0	100%
Matmul	2	yes	iter	3	0	8	0	100%
TOTAL								98.88%

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Auto-scope results

	Description			LLVM Results				
Benchmarks	**#tasks**	**nested tasks**	**method**	**shared**	**private**	**firstprivate**	**undefined**	**(%) success**
Alignment	1	no	iter	2	4	14	0	**100%**
FFT	41	no	rec	102	0	140	0	**100%**
Fib	2	no	rec	2	0	3	0	**100%**
Health	2	yes	iter&rec	1	1	3	0	**100%**
Floorplan	1	no	iter&rec	3	1	9	2	**86.66%**
Nqueens	1	no	iter&rec	2	0	4	0	**100%**
Sort	9	yes	rec	27	0	10	0	**100%**
SparseLU	4	yes	iter	4	3	11	0	**100%**
UTS	2	no	iter&rec	2	1	3	0	**100%**
Cholesky	4	no	iter	4	0	12	0	**100%**
Saxpy	2	yes	iter	4	0	3	0	**100%**
Matmul	2	yes	iter	3	0	8	0	**100%**
TOTAL								**98.88%**

Variables used in system calls, and not defined in the reachable code

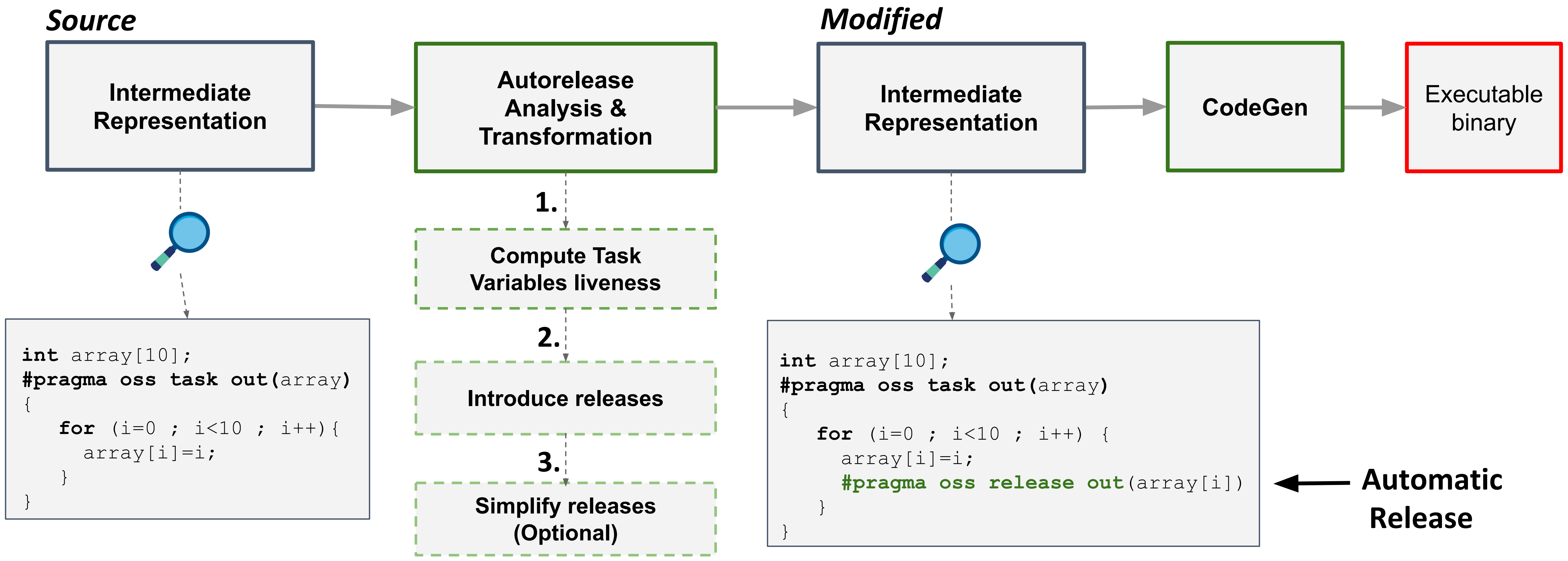
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outline

- Introduction to OmpSs-2
- Proposed algorithms for programmability and performance
    - Auto-scope + Evaluation
    - **Auto-release + Evaluation**
- Implementation
- Discussion

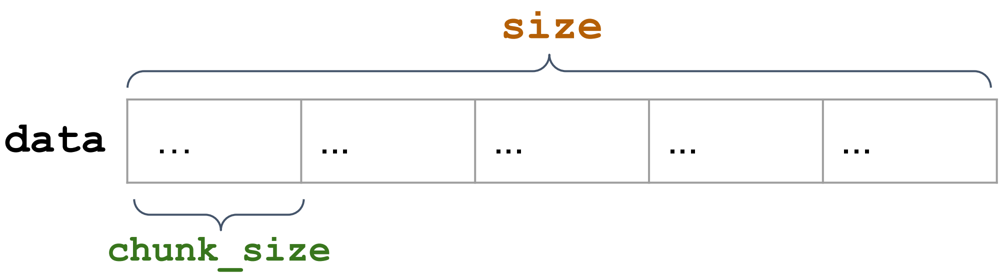
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

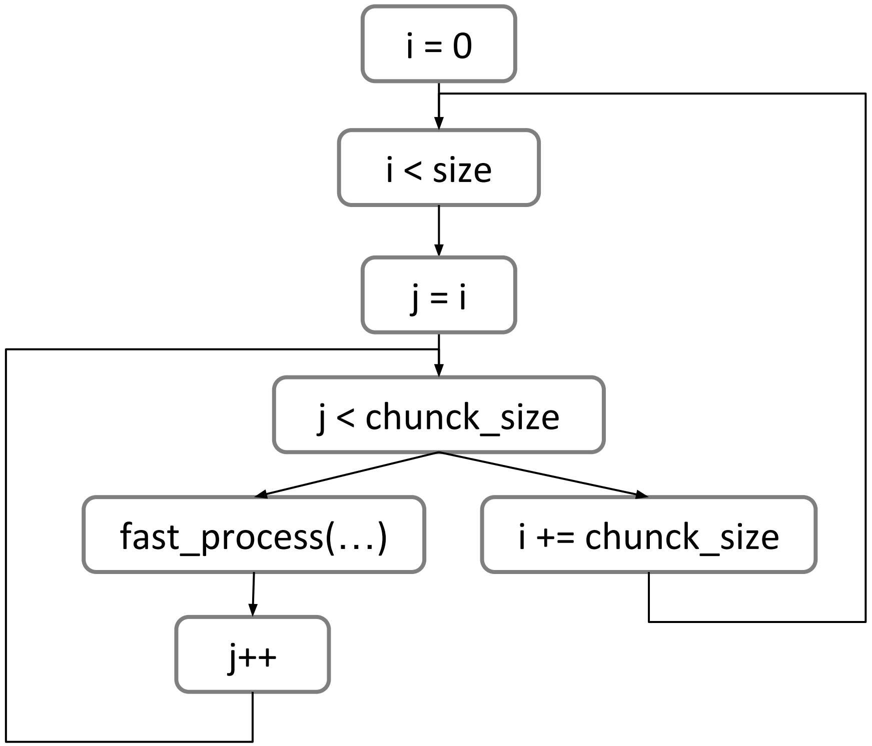
Autorelease overview



Autorelease by example

```
#pragma oss task out(data[0; size])
for (int i = 0; i < size; i += chunk_size) {
    // data[i;chunk_size] is used
    for (int j = i; j < chunk_size; j++) {
        fast_process(&data[j]);
        // data[j] is never used again
        // i and j are used in the
        // respective loop increments
    }
}
```

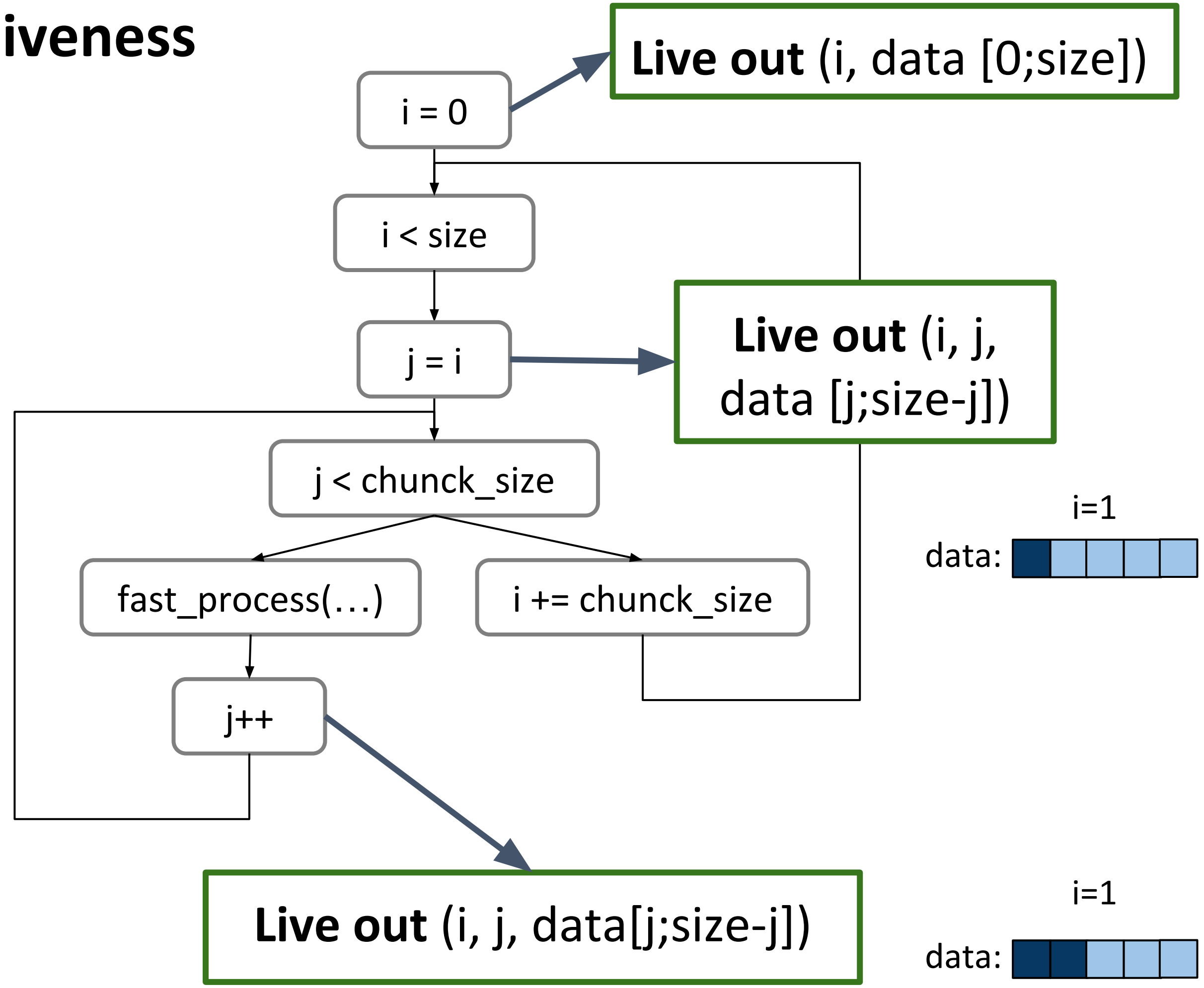




Autorelease by example

Compute liveness

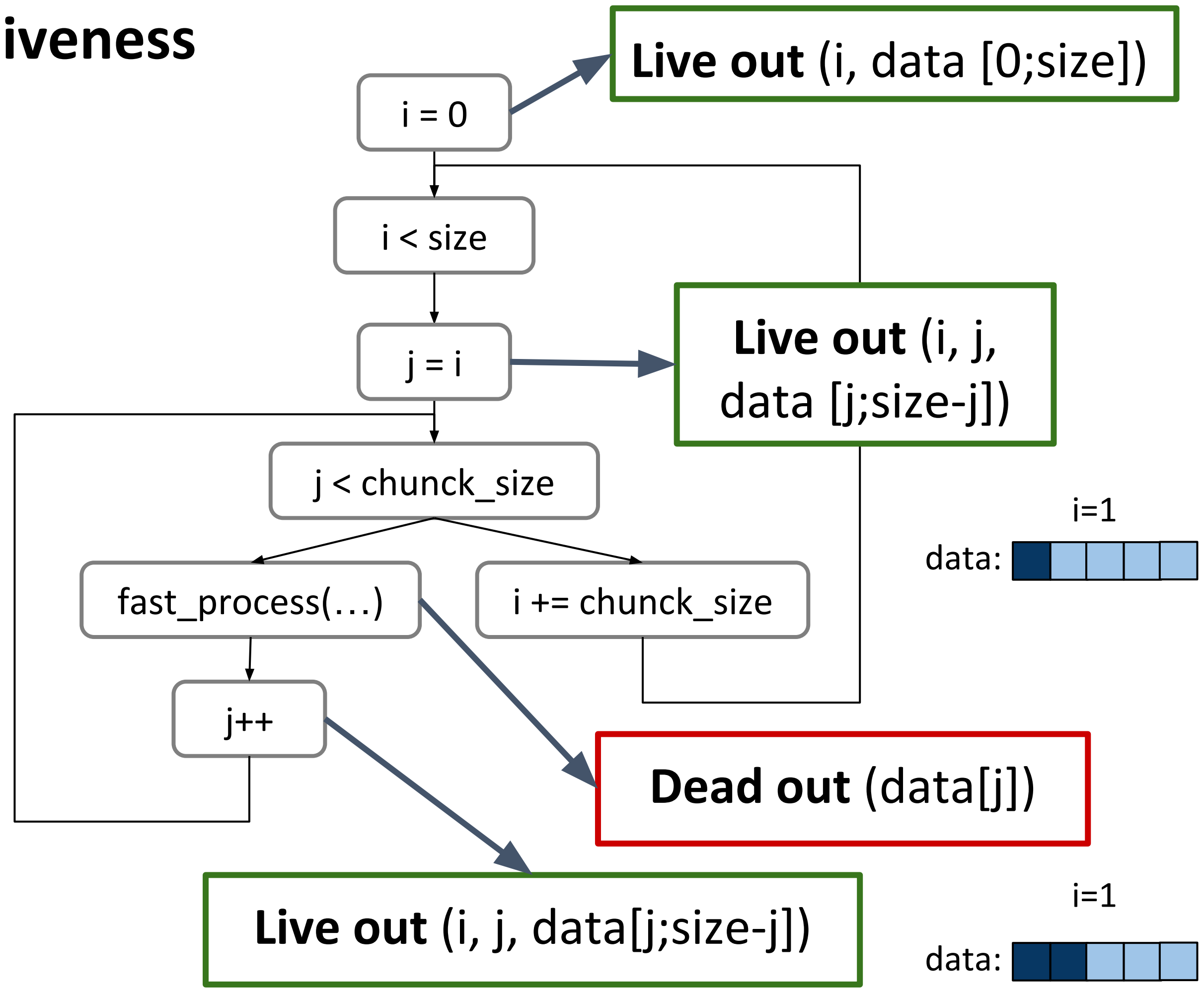
```
#pragma oss task out(data[0; size])
for (int i = 0; i < size; i += chunk_size) {
   // data[i;chunk_size] is used
   for (int j = i; j < chunk_size; j++) {
      fast_process(&data[j]);
      // data[j] is never used again
      // i and j are used in the
      // respective loop increments
   }
}
```



Autorelease by example

Compute liveness

```
#pragma oss task out(data[0; size])
for (int i = 0; i < size; i += chunk_size) {
   // data[i;chunk_size] is used
   for (int j = i; j < chunk_size; j++) {
      fast_process(&data[j]);
      // data[j] is never used again
      // i and j are used in the
      // respective loop increments
   }
}
```



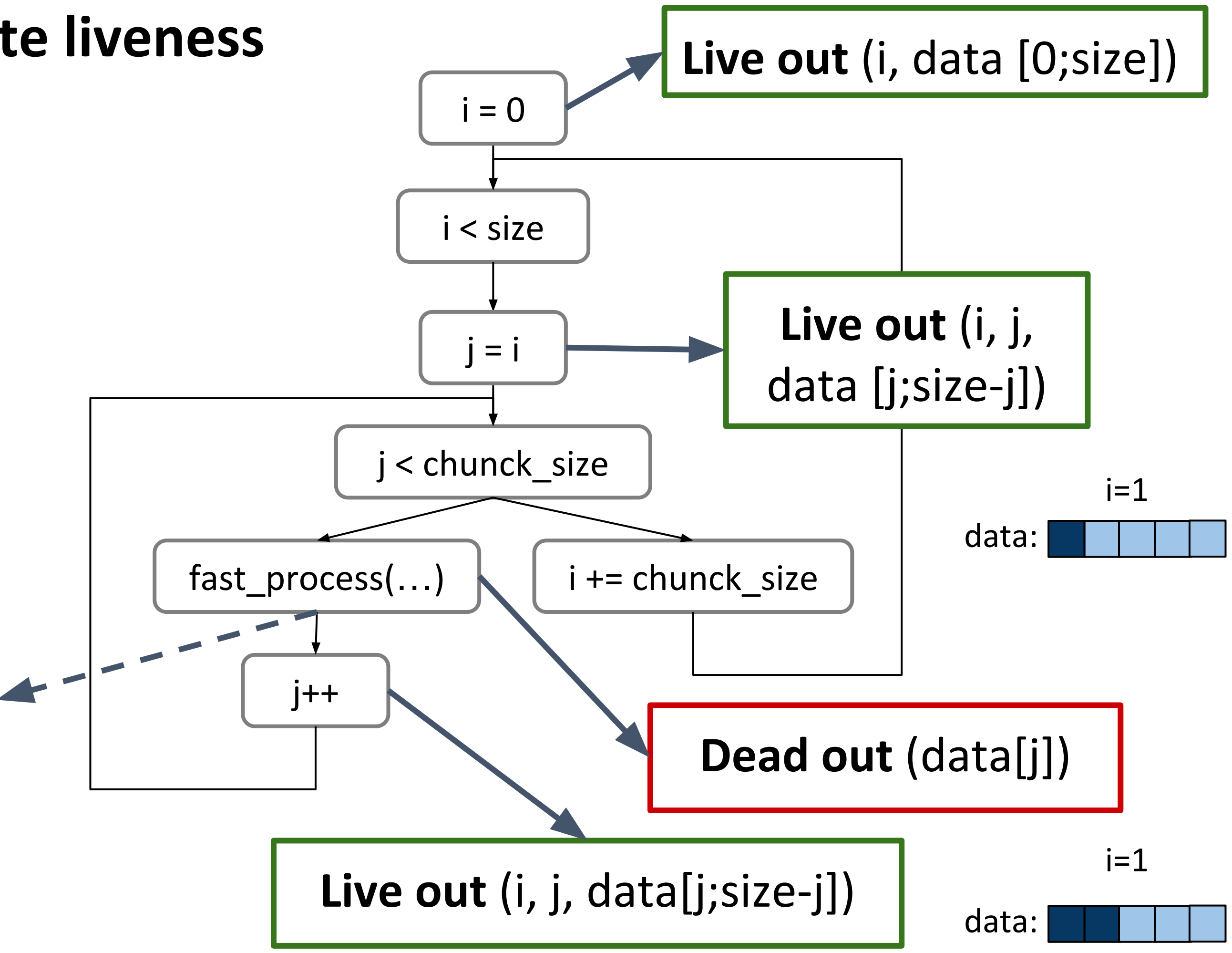
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Autorelease by example

Compute liveness

```
#pragma oss task out(data[0; size])
for (int i = 0; i < size; i += chunk_size) {
   // data[i;chunk_size] is used
   for (int j = i; j < chunk_size; j++) {
      fast_process(&data[j]);
      // data[j] is never used again
      // i and j are used in the
      // respective loop increments
      #pragma oss release (data[j]);
   }
}
```

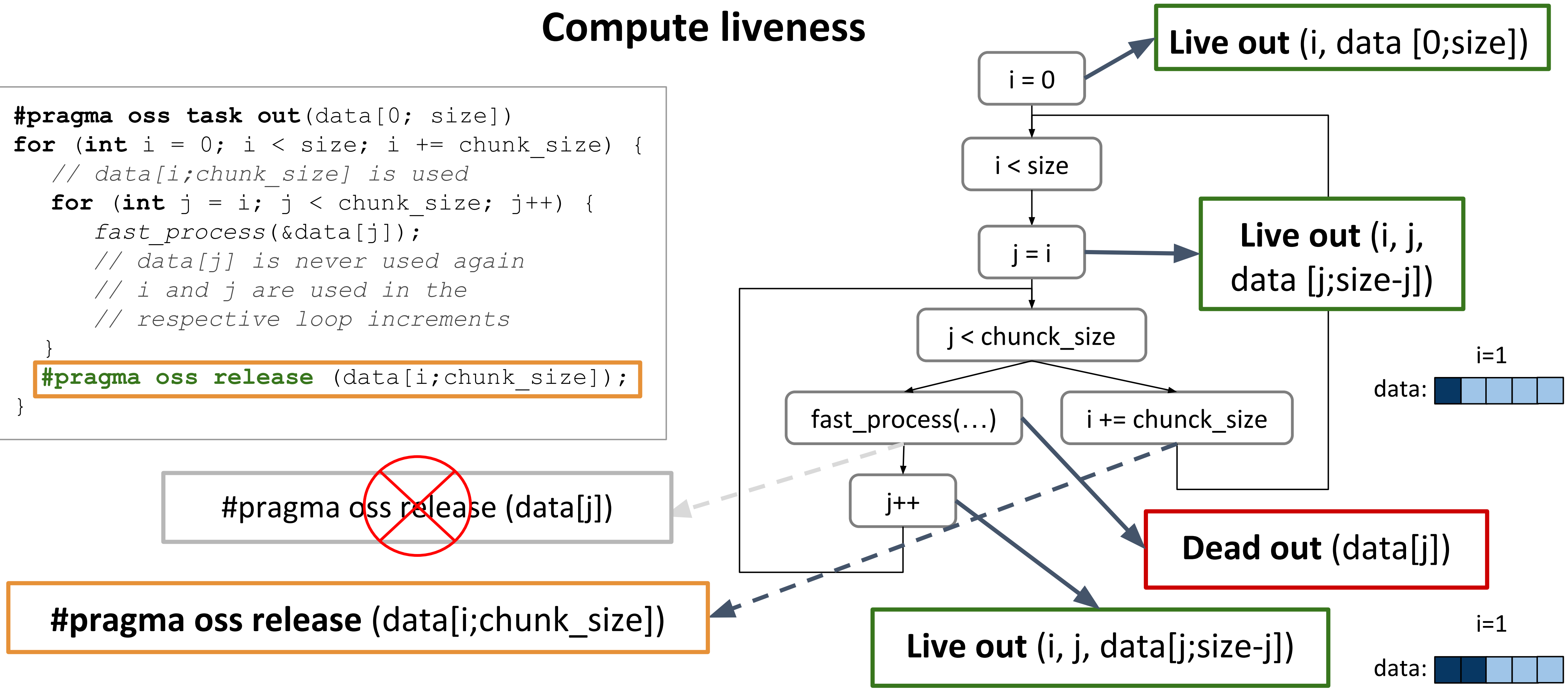
#pragma oss release (data[j])



Autorelease by example

Compute liveness

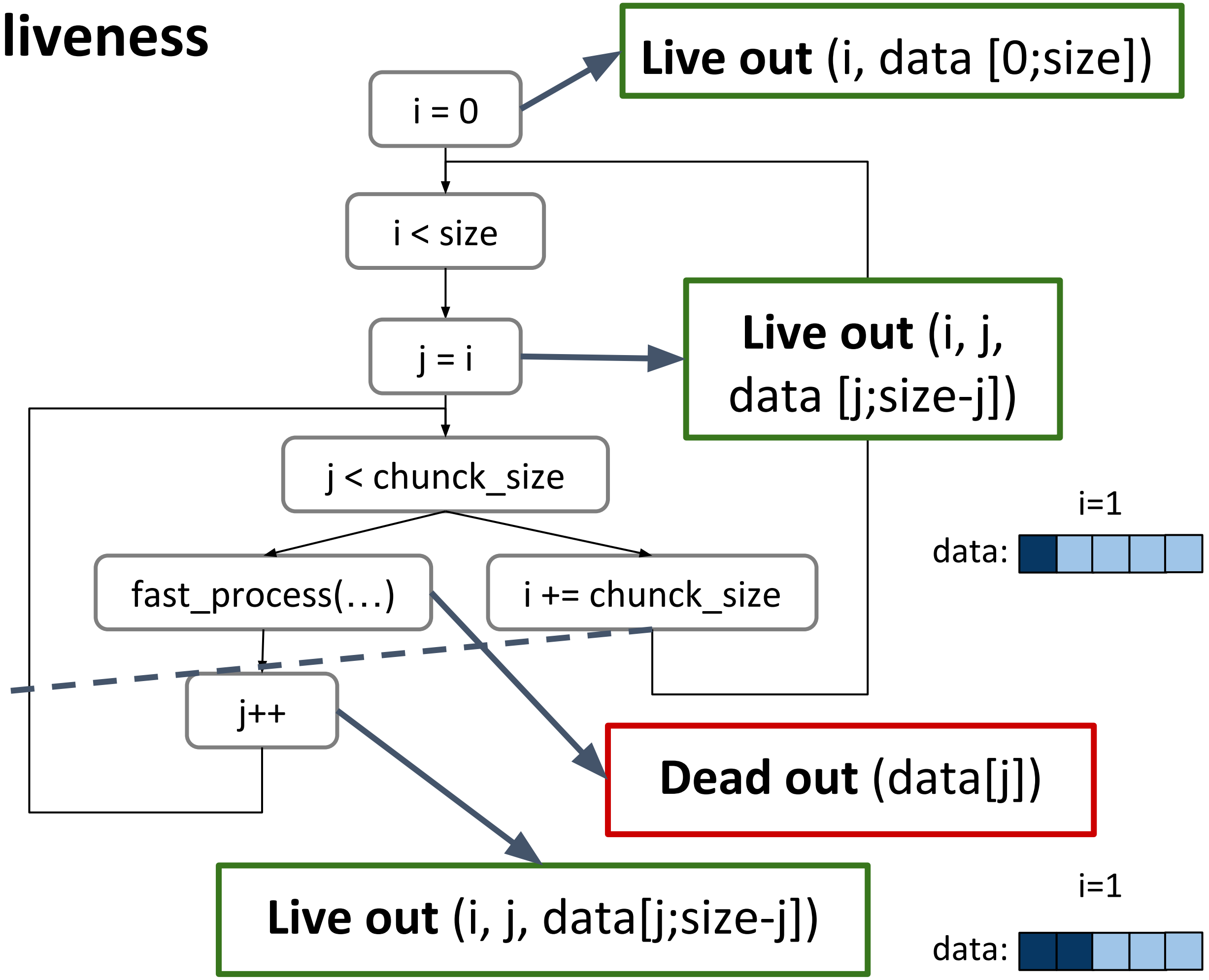
```
#pragma oss task out(data[0; size])
for (int i = 0; i < size; i += chunk_size) {
   // data[i;chunk_size] is used
   for (int j = i; j < chunk_size; j++) {
      fast_process(&data[j]);
      // data[j] is never used again
      // i and j are used in the
      // respective loop increments
   }
   #pragma oss release (data[i;chunk_size]);
}
```



Autorelease by example

Compute liveness

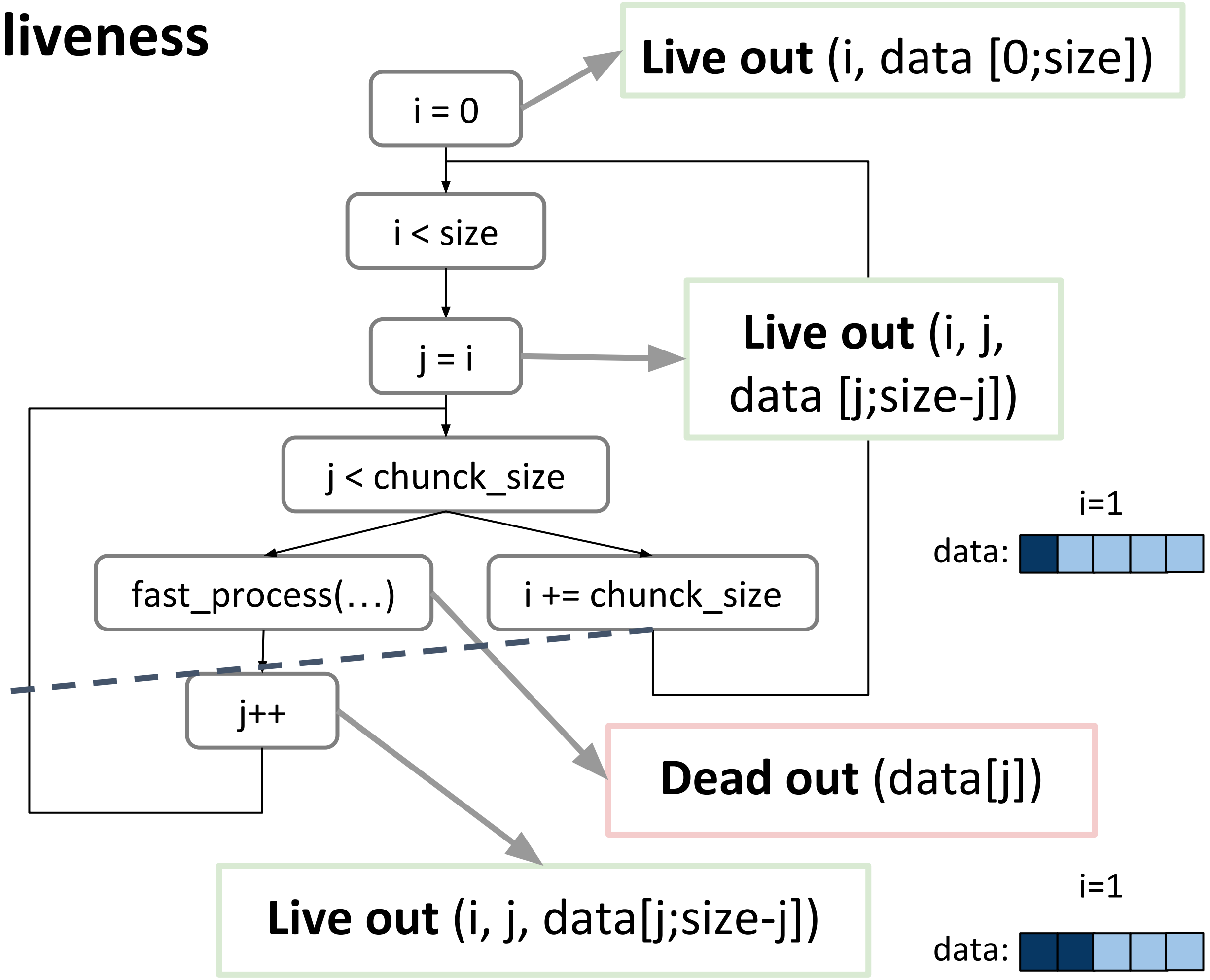
```
#pragma oss task out(data[0; size])
for (int i = 0; i < size; i += chunk_size) {
   // data[i;chunk_size] is used
   for (int j = i; j < chunk_size; j++) {
      fast_process(&data[j]);
      // data[j] is never used again
      // i and j are used in the
      // respective loop increments
      #pragma oss release (data[j]);
   }
   #pragma oss release (data[i;chunk_size]);
}
```



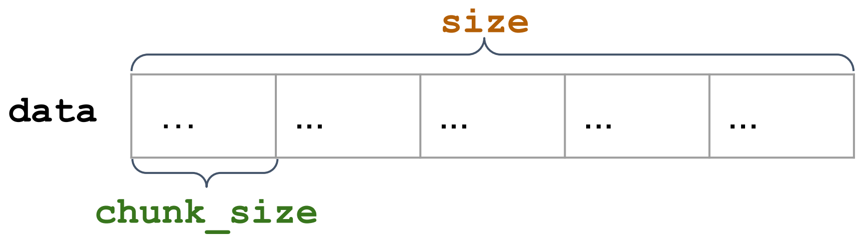
Autorelease by example

Compute liveness

```
#pragma oss task out(data[0; size])
for (int i = 0; i < size; i += chunk_size) {
   // data[i;chunk_size] is used
   for (int j = i; j < chunk_size; j++) {
      fast_process(&data[j]);
      // data[j] is never used again
      // i and j are used in the
      // respective loop increments
      #pragma oss release (data[j]);
   }
  #pragma oss release (data[i;chunk_size]);
}
```



Autorelease evaluation




```
int data[size];

#pragma oss task out(data[0; size]) label(T1)
{
  for (int i = 0; i < size; i += chunk_size) {
    for (int j = i; j < chunk_size; j++)
      process(&data[j]);
       //Release within loop (data[j])
    }
    //Release outside loop (data[i;chunk_size])
  }
  for (int i = 0; i < size; i += chunk_size) {
    #pragma oss task in(data[i; chunk_size]) label(T2)
    {
      slow_process(&data[i], chunk_size);
    }
  }
}
```

	Variables	
process	**Super fast process**	**Fast process**
execution time	100 **us**	1000 **us**
size	200,000	20,000
chunk_size	10,000	1,000

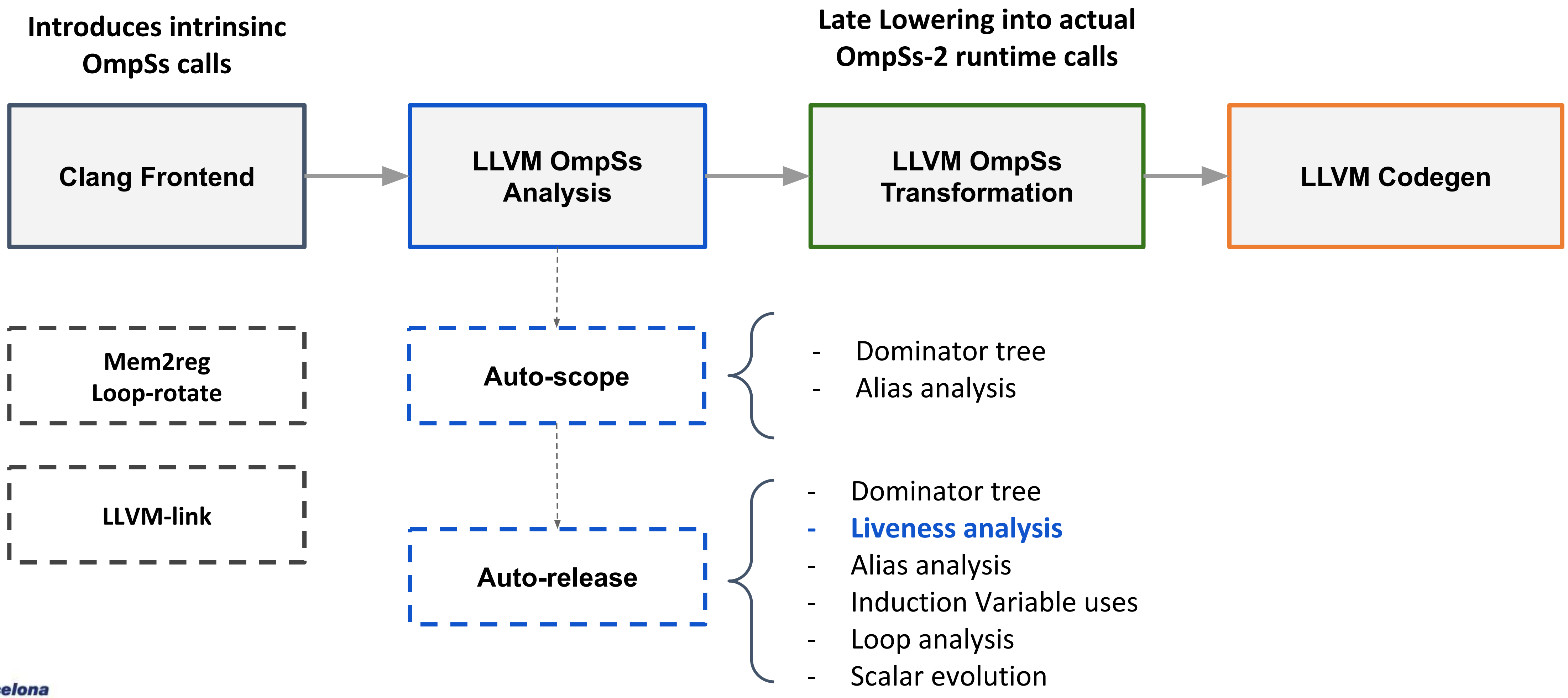
	Execution time (us)	
process	**Super fast process**	**Fast process**
No release	**46**,747,405	**36**,233,344
Release within loop	**40**,607,605	**26**,570,543
Release outside loop	**35**,894,357	**26**,533,354

Outline

- Introduction to OmpSs-2
- Proposed algorithms for programmability and performance
  - Auto-scope + Evaluation
  - Auto-release + Evaluation
- **Implementation**
- Discussion

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

LLVM implementation



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outline

- Introduction to OmpSs-2
- Proposed algorithms for programmability and performance
  - Auto-scope + Evaluation
  - Auto-release + Evaluation
- Implementation
- **Discussion**

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Applicability for OpenMP

- The automatic scope of variables in task constructs can be applied for OpenMP
- Some features of OmpSs-2 are already in the newest OpenMP specifications.
  - **`commutative`** clause: has been introduced into the OpenMP 5.0 as *mutexinoutset*.
  - **`concurrent`** clause: a first preview of the OpenMP 5.1 introduces the *inoutset* clause, with the same behaviour.
- OmpSs-2 forces parent tasks to cover the dependencies of children tasks with either regular dependencies or ***weak dependencies***. This restriction could be applied to OpenMP if this model is to be used in critical real-time systems.
- Others features such as the **`release`** clause does not exists on OpenMP but can be used for other models with similar functionality, as *DepSpawn.*

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusions

As a conclusion, this works tackles:

1. **Programmability:** Auto scoping the data sharings help users to hide complexity.
2. **Correctness:** Avoid possible human errors that sometimes can be difficult to track.
3. **Performance:** Improve performance introducing autorelease of dependencies.

However, remains as a future work to:

1. Simplify the release of dependencies (join contiguous accesses).
2. Automatically determine the dependency clauses (*auto-deps*).
3. Evaluate the possibilities of the OpenMP `detach` clause.

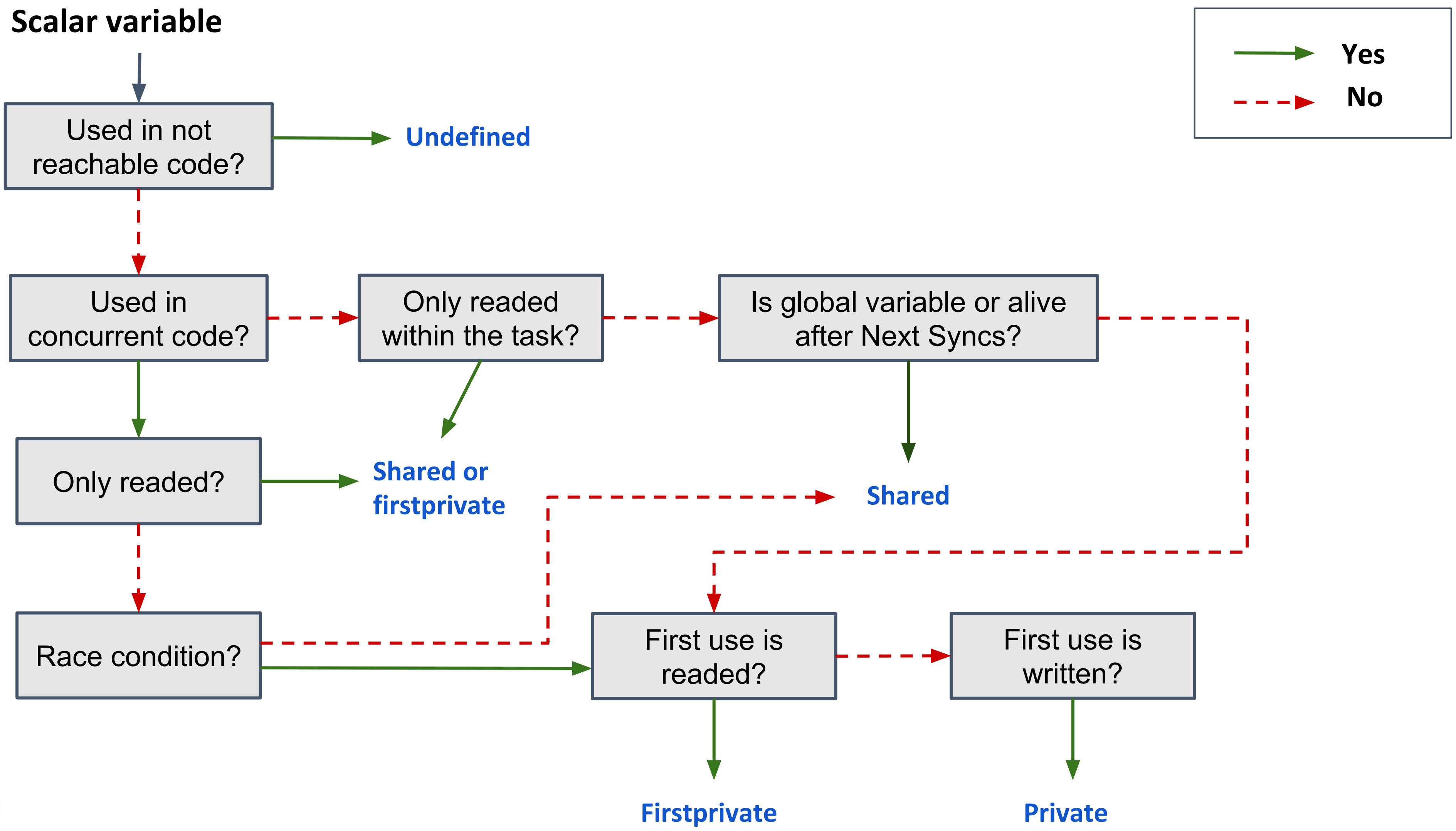
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thank you

adrian.munera@bsc.es

Auto-scope rules for scalar variables



Barcelona Supercomputing Center
Centro Nacional de Supercomputación