

Task-Based Programming Languages

Hans Vandierendonck

March 30, 2011

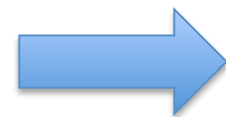
Introduction

- **Tasks + dependencies - dataflow style**
 - Good performance on irregular dependencies (hard to do manually)
 - Sequential-like programming model
 - Complete footprints facilitate execution on distributed memory systems (Cell B.E., GPUs, ...)
 - Patterns to keep in mind:
 - Scientific applications calling BLAS, LAPACK kernels
 - Pipelines

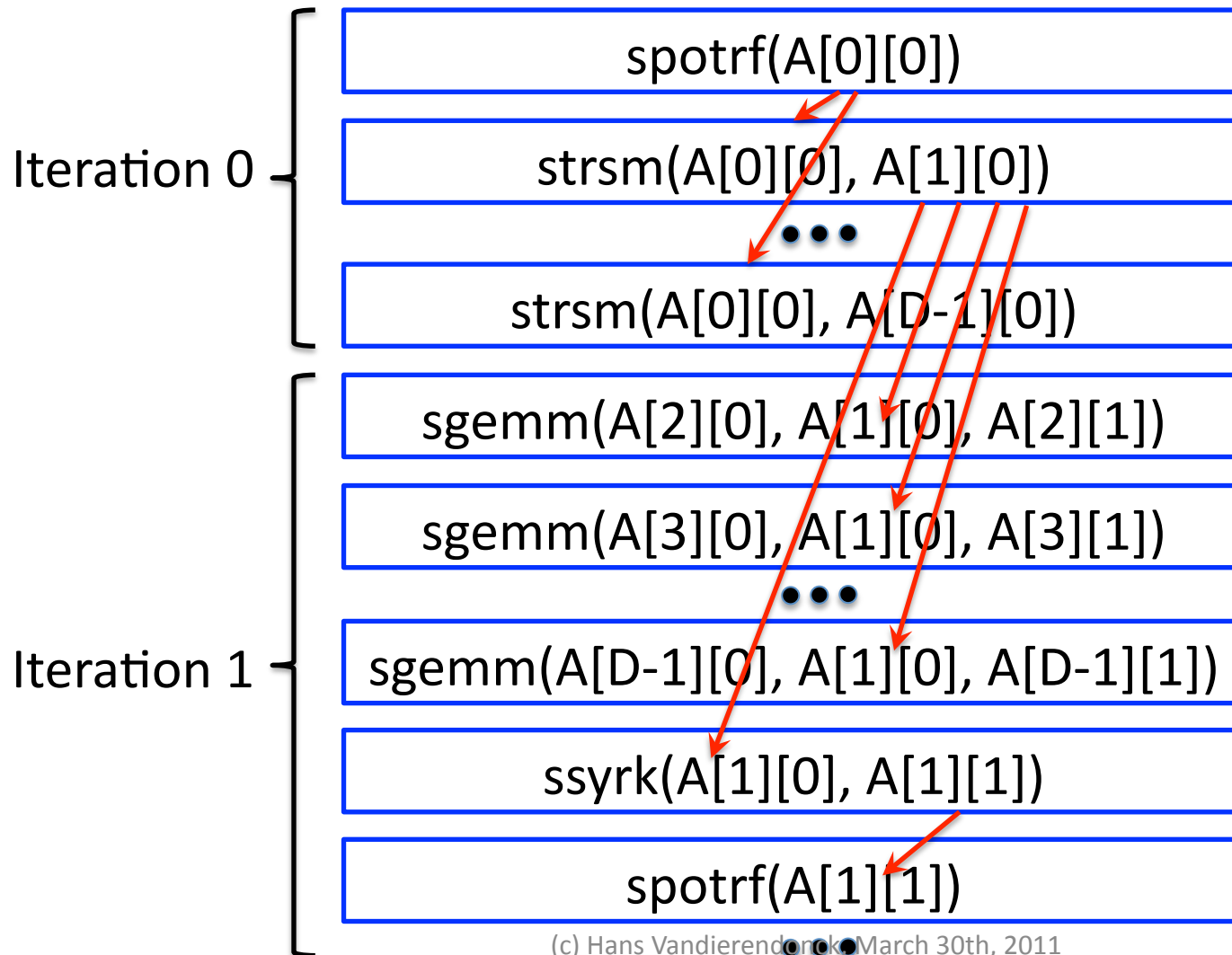
Example: Cholesky Decomposition

- Blocked implementation
- BLAS kernels operate on a $B \times B$ matrix
- Where is the parallelism?

```
/* A[D][D] is a hypermatrix of BxB blocks */
for (int j = 0; j<D; j++) {
    for (int k = 0; k<j; k++)
        for (int i = j+1; i<D; i++)
            sgemm(A[i][k], A[j][k], A[i][j]);
    for (int i = 0; i<j; i++)
        ssyrk(A[j][i], A[j][j]);
    spotrf(A[j][j]);
    for (int i = j+1; i<D; i++)
        strsm(A[j][j], A[i][j]);
}
```



Tracing the Kernels



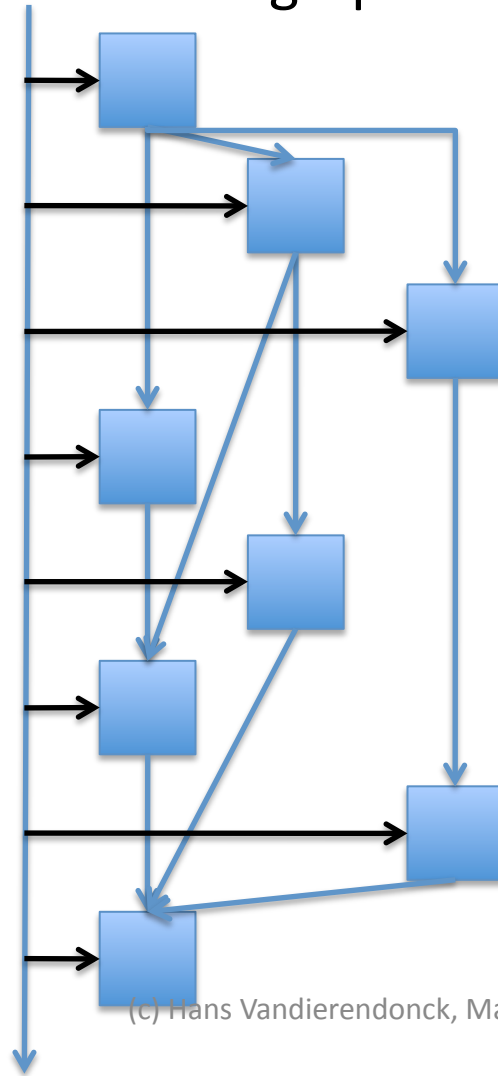
There is parallelism between kernels!

Principle: Dynamically Extracting Task Graph

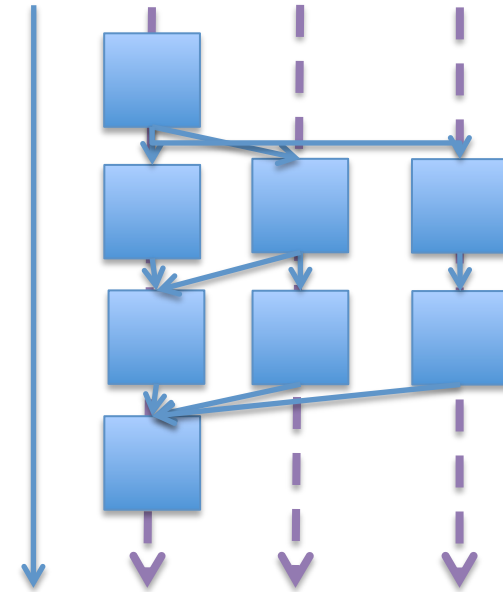
Sequential execution



Task graph



Parallel execution



THE SMPSS PROGRAMMING MODEL

SMPSS: SMP SuperScalar

- **Targets HPC**
 - Tasks are computational kernels (BLAS, LAPACK)
 - Blocked algorithms (hypermatrix)
- **Simple programming model**
 - #pragma based
 - Task arguments are annotated by programmer
 - Look-and-feel of a sequential language
- **Runtime system**
 - Finds parallelism between tasks
 - Correctly orders execution of tasks
- **Material based on:**
 - Perez, J.M., Badia, R.M. and Labarta, J. “A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures”. In: Intl. Conf. on Cluster Computing, 2008.

SMPSS: Tasks

- **Tasks are procedures**
 - **Leaf** tasks: tasks do not spawn other tasks
- **Annotate arguments on task declarations**
 - Base address + size (bytes) + dependency type
 - `input(arg)`: input argument (read-only)
 - `output(arg)`: output argument (write-only)
 - `inout(arg)`: input/output argument (read-write)
 - Arguments should not contain pointers
- **Example:**

```
#pragma css task input(a[NB][NB], NB) inout(c[NB][NB])  
void sgemm_t(float a[NB][NB], float c[NB][NB], long NB);
```

SMPSS: Main code

- Main code simply calls tasks in a sequential manner
- Example: Cholesky factorization

```
#pragma css start
for (int j = 0; j<N; j++) {
    for (int k = 0; k<j; k++)
        for (int i = j+1; i<N; i++)
            sgemm_t(A[i][k], A[j][k], A[i][j]);
    for (int i = 0; i<j; i++)
        ssyrk_t(A[j][i], A[j][j]);
    spotrf_t(A[j][j]);
    for (int i = j+1; i<N; i++)
        strsm_t(A[j][j], A[i][j]);
}
#pragma css finish
```

SMPSS: Code Transformation

- SMPSS compiler replaces task calls by calls to the runtime system
 - Task to launch
 - Arguments (base address + size + dependency type)

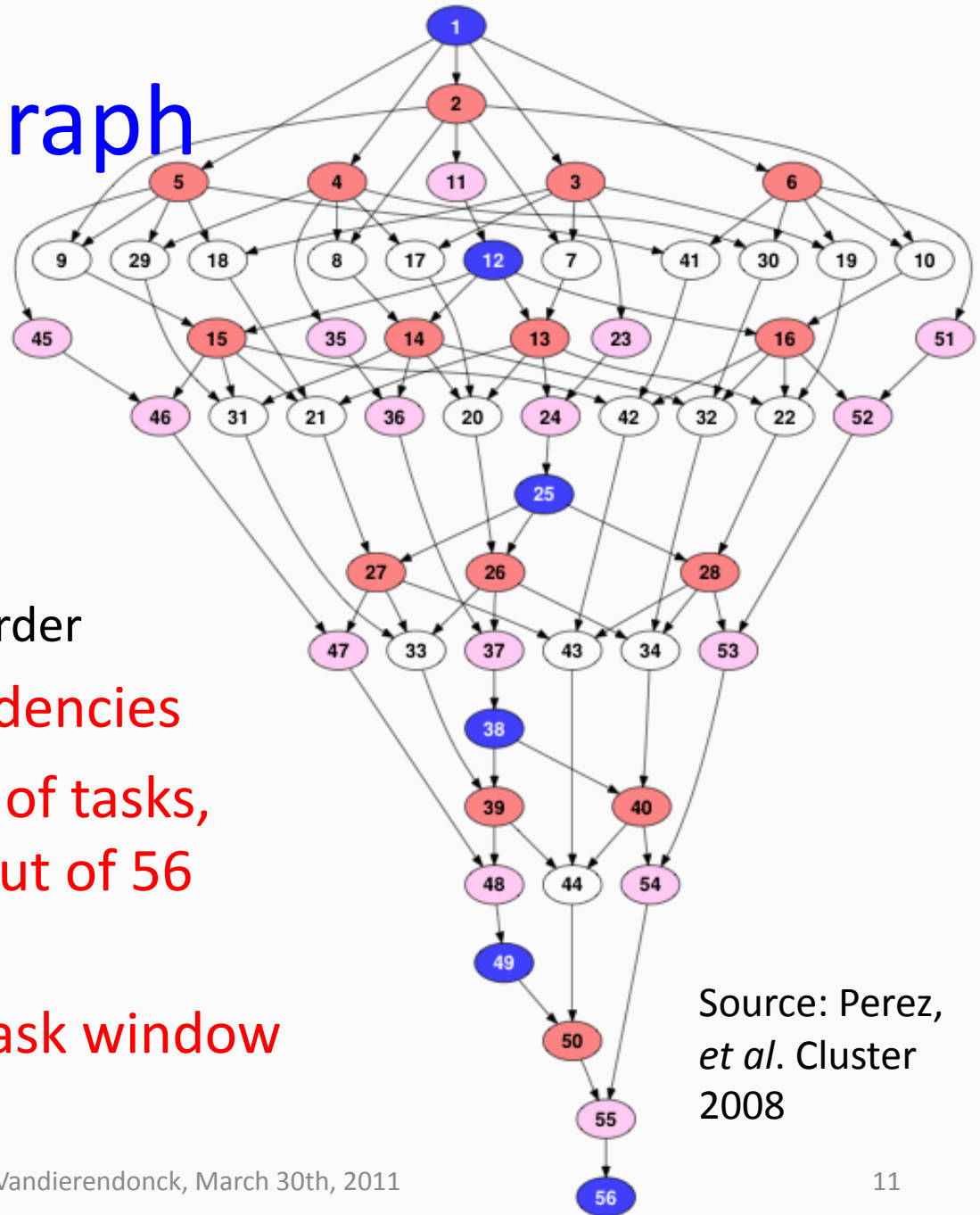
```
ssyrk_t(A[j][i], A[j][j]);
```

```
css_parameter_t parameters [3];  
parameters [0].flags = CSS_IN_DIR;  
parameters [0].size = (NB) * (NB) * sizeof(float);  
parameters [0].address = A[j][i];  
parameters [1].flags = CSS_INOUT_DIR;  
parameters [1].size = (NB) * (NB) * sizeof(float);  
parameters [1].address = A[j][j];  
parameters [2].flags = CSS_IN_SCALAR_DIR;  
parameters [2].size = sizeof(long int);  
parameters [2].address = &(NB);  
css_addTask(ssyrk_t, CSS_NO_FLAG, 3, parameters);
```

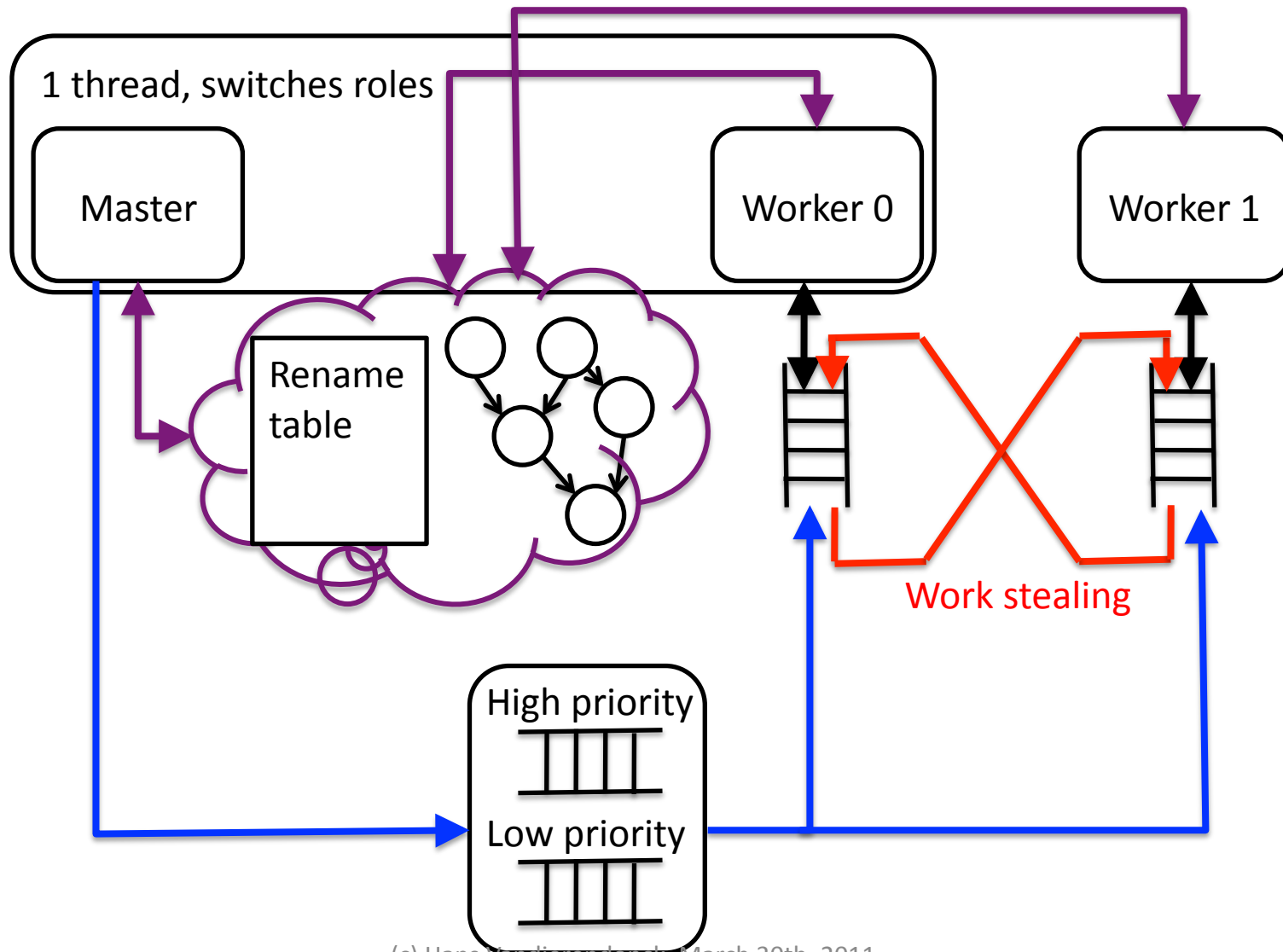
(c) Hans Vandierendonck, March 30th, 2011

SMPSS: Task Graph

- 6x6 cholesky factorization
- Nodes are tasks
 - Color: task type
 - Number: invocation order
- Edges are true dependencies
- Important reordering of tasks, e.g. execute task 51 out of 56 already after task 6
- Importance of large task window



SMPSS: Runtime System



DEPENDENCY RESOLUTION IN SMPSS

Ingredients of a Task-Based Language

- **Definition of tasks and their dependencies**
 - Tasks are procedures
 - Name-based vs data-based dependencies
- **Dependency resolution**
 - Build task graph
 - Determine when tasks are ready to execute
- **Dependency-aware scheduling**
 - Schedule ready tasks on workers
 - Potential optimizations by leveraging task dependencies!
 - E.g. increasing parallelism, locality optimization

Task Dependencies

- Name-based dependencies
 - “Task T must execute after Task S”
 - Main examples: OpenCL, StarPU

```
/* Assumes an out-of-order command queue */
cl_uint num_events_in_waitlist = 2;
cl_event event_waitlist[2];

err = clEnqueueReadBuffer(queue, buffer0, CL_FALSE /* non-blocking */,
                          0,0,0, NULL, &event_waitlist[0]);
err = clEnqueueReadBuffer(queue, buffer1, CL_FALSE /* non-blocking */,
                          0,0,0, NULL, &event_waitlist[1]);
/* last read buffer waits on previous two read buffer events */
err = clEnqueueReadBuffer(queue, buffer2, CL_FALSE /* non-blocking */,
                          0,0,num_events_in_waitlist, event_waitlist, NULL);
```

Name-Based Task Dependencies

- Difficult in irregular applications

```
/* Assumes an out-of-order command queue */
cl_uint num_events_in_waitlist = 2;
cl_event event_waitlist[2];

err = clEnqueueReadBuffer(queue, buffer0, CL_FALSE /* non-blocking */,
                          0,0,0, NULL, &event_waitlist[0]);

if( some_condition ) {
    err = clEnqueueReadBuffer(queue, buffer1, CL_FALSE /* non-blocking */,
                              0,0,0, NULL, &event_waitlist[1]);
}

/* now do we wait on one or two events? */
err = clEnqueueReadBuffer(queue, buffer2, CL_FALSE /* non-blocking */,
                          0,0,num_events_in_waitlist, event_waitlist, NULL);
```

Data-Based Task Dependencies

- **Data-based dependencies**
 - Specify **memory footprint** of each task
 - Runtime system **infers** dependencies
 - Spawn task S writing to variable “A”
 - Spawn task T reading from variable “A”
 - Therefore, T must execute after S
- **Specifying footprints**
 - Single-level data structures (contain no pointers)
 - E.g. leaves in a hypermatrix
 - Compiler helps to compute footprint (approximately)
 - E.g. OoOJava
 - Type system allows to specify left/right subtrees in a tree
 - E.g. Deterministic Parallel Java

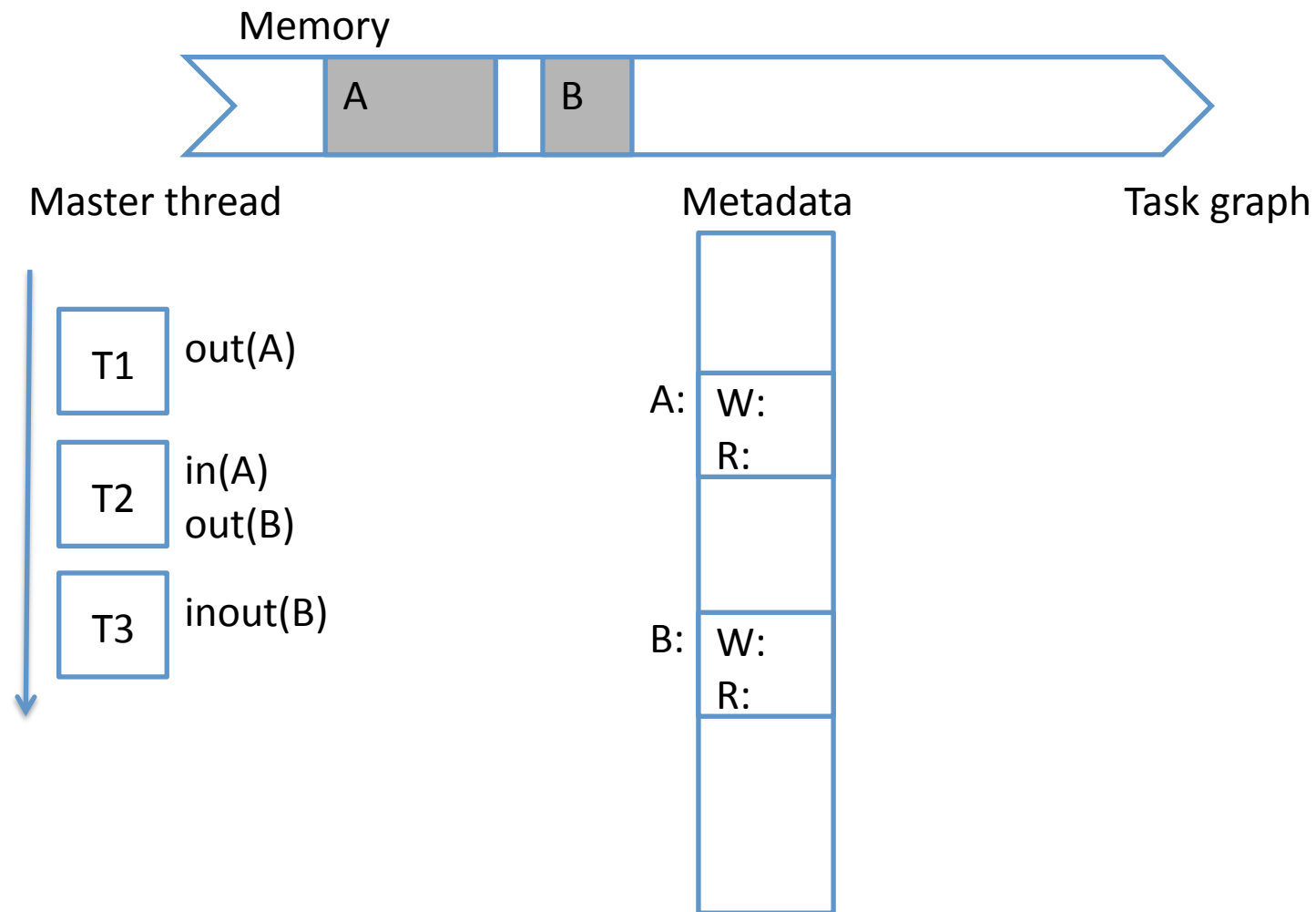
Dependency Types

- **RAW = read-after-write**
 - A.k.a. true dependency
 - Task T has a RAW dependency on task S
 - T must execute after S because it needs the data
- **WAR = write-after-read**
 - A.k.a. anti dependency
 - Task T has a WAR dependency on task S
 - T must wait until S is finished reading the common argument
 - Alternative: we let T write into a fresh copy of the argument
 - Requires runtime system support ; discussed later
 - Key optimization also in dependency handling in superscalar processors
- **WAW = write-after-write**
 - A.k.a. output dependency
 - Same treatment as WAR dependencies

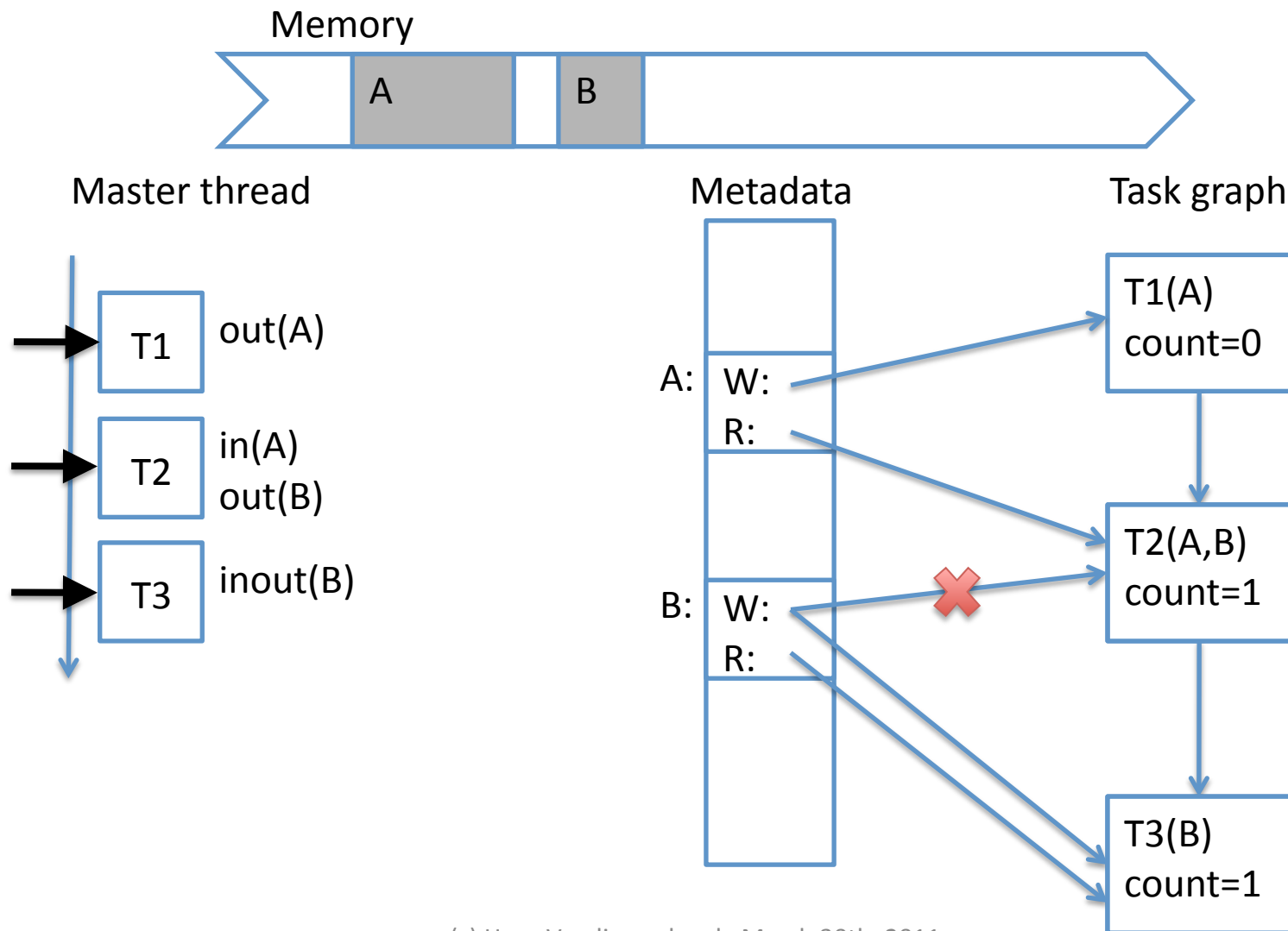
SMPSS: Dependency Resolution

- Based on the *representant* of an argument
 - The representant is the base address of an argument
 - The representant is the address passed to a task
 - In principle object-based (argument=object)
- Dependency resolution based on matching representants
 - Map representant to last writing/reading tasks
 - If the new task reads an argument, wait on last writing task
 - If the new task writes an argument, wait on all writing and reading tasks
 - Implementation uses a hash table to track last writing/reading tasks for all representants
- Some tricks to work around object-based (iff no renaming, see later)

SMPSS: Building the Task Graph



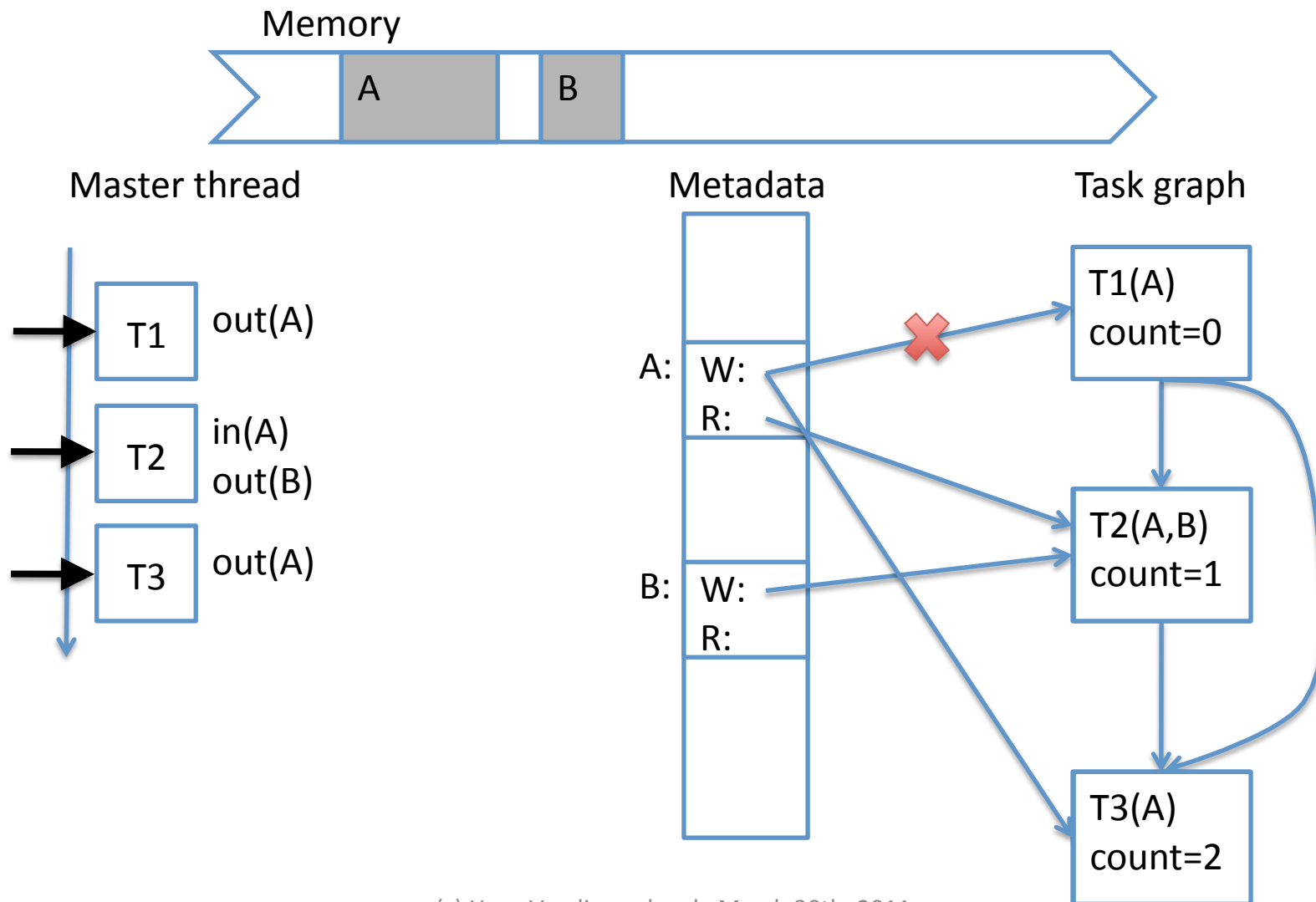
SMPSS: Building the Task Graph



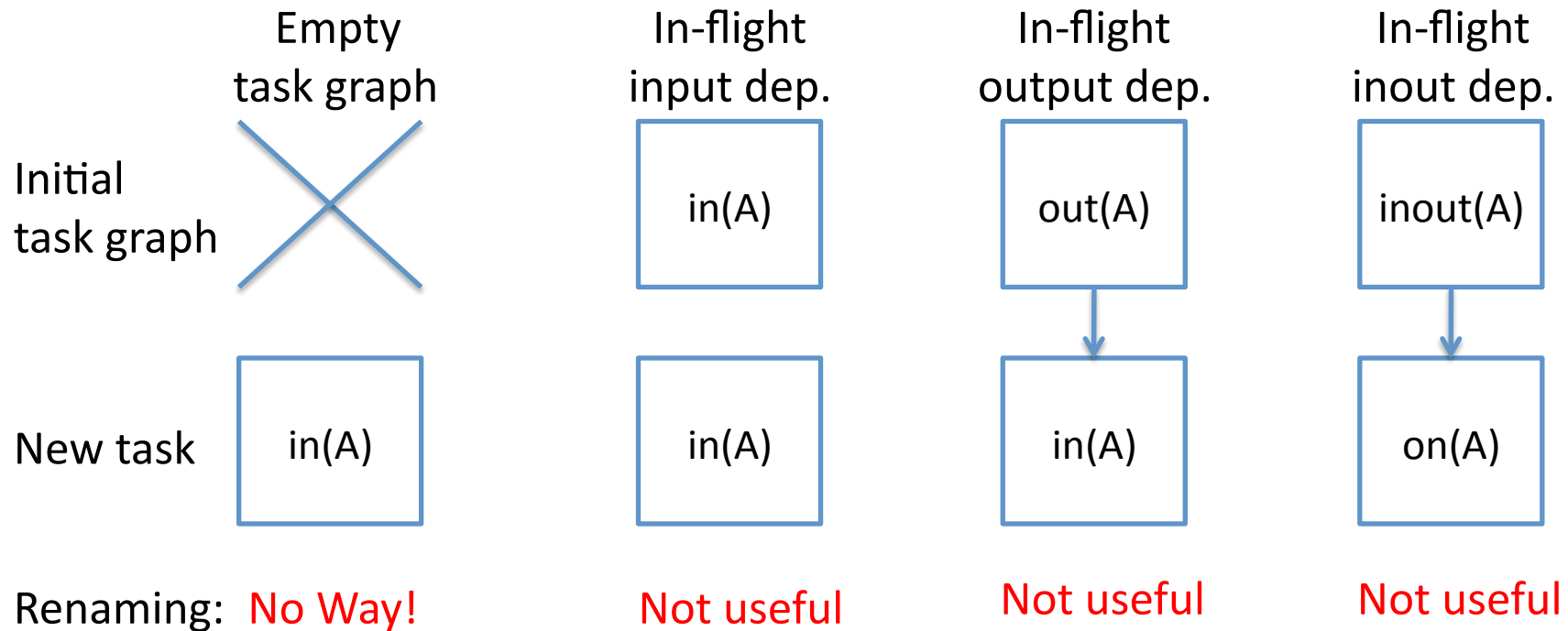
SMPSS: Increasing Parallelism with Renaming

- WAR and WAW dependencies limit parallelism
- Renaming: reserve new memory area for the argument to operate on in future tasks
 - We know base address + size
 - Reuse hash table to store “current” version of argument
- Copy-back when programmer requests access to data in master thread

Example: WAR Without Renaming

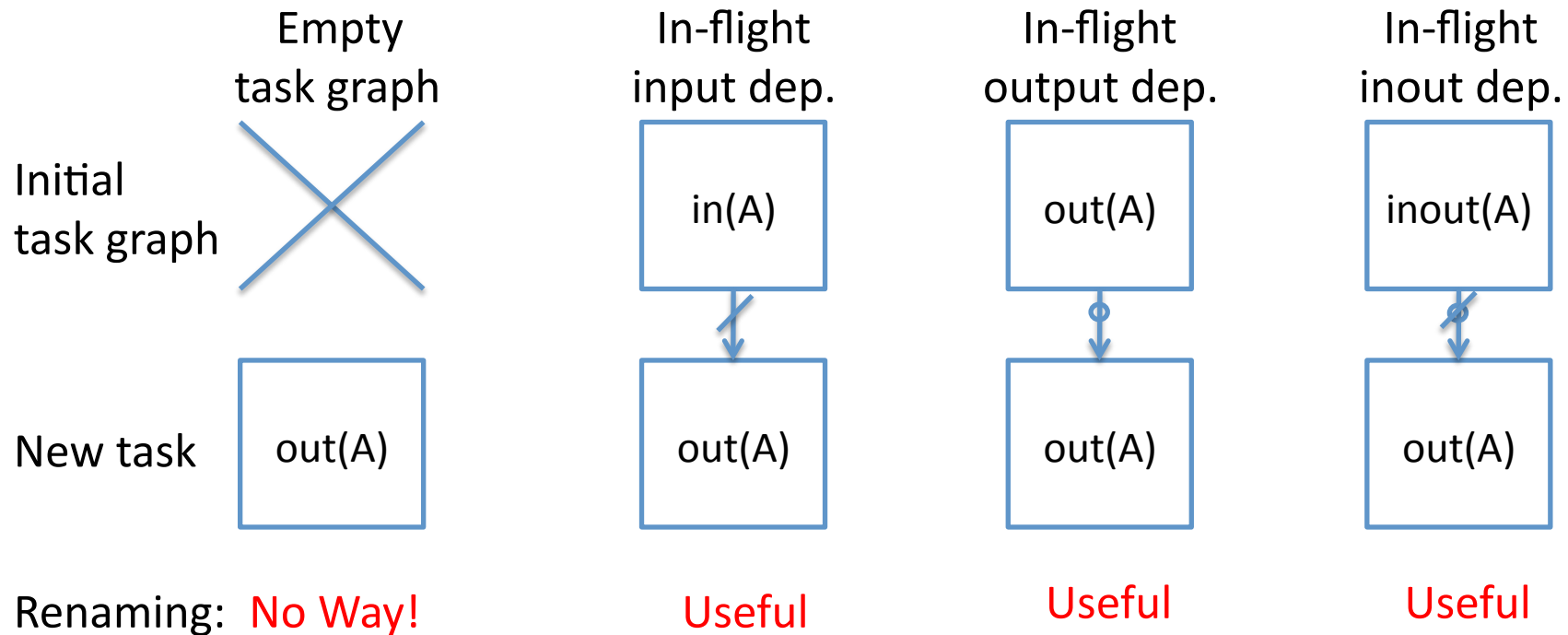


Renaming: Arguments of Type “input”



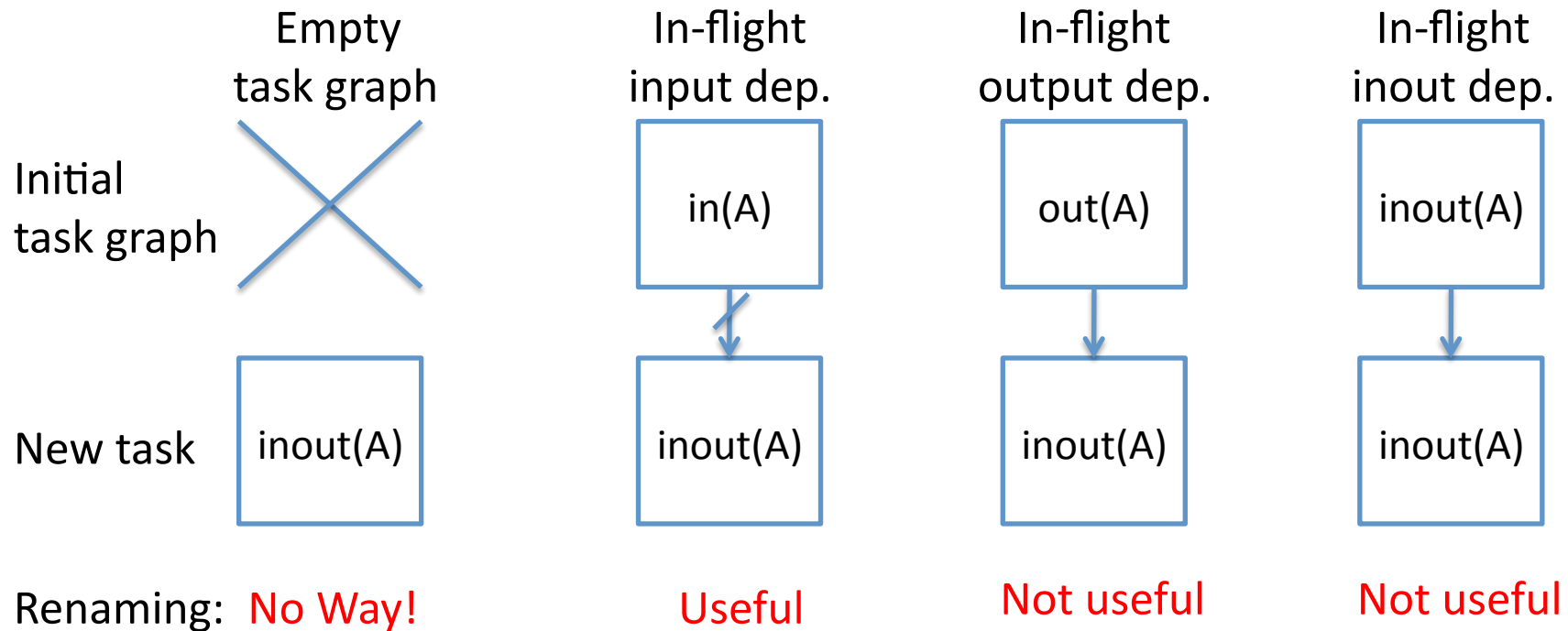
- Input dependencies never require renaming

Renaming: Arguments of Type “output”



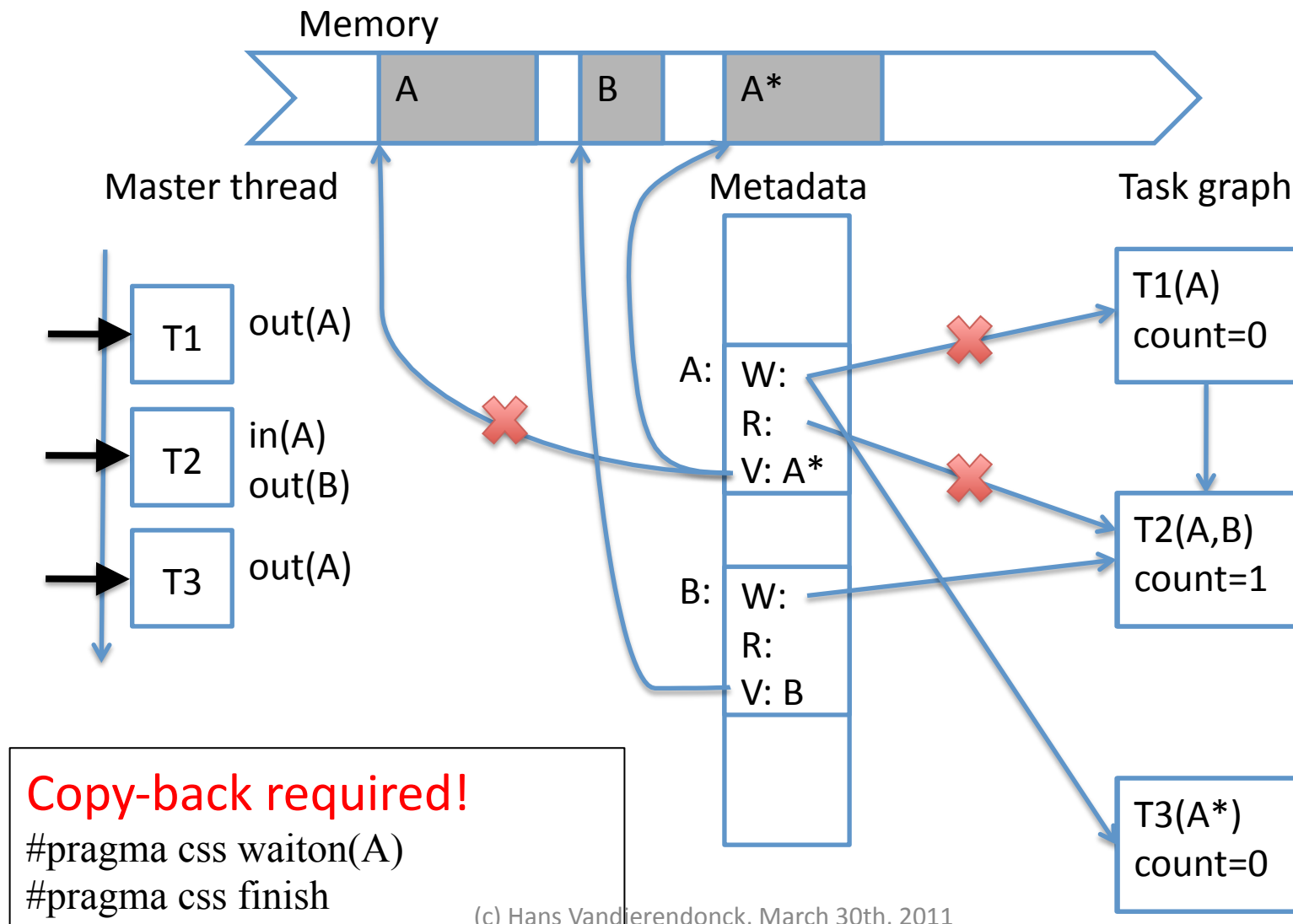
- Always rename if a prior task touches the same data

Renaming: Arguments of Type “inout”

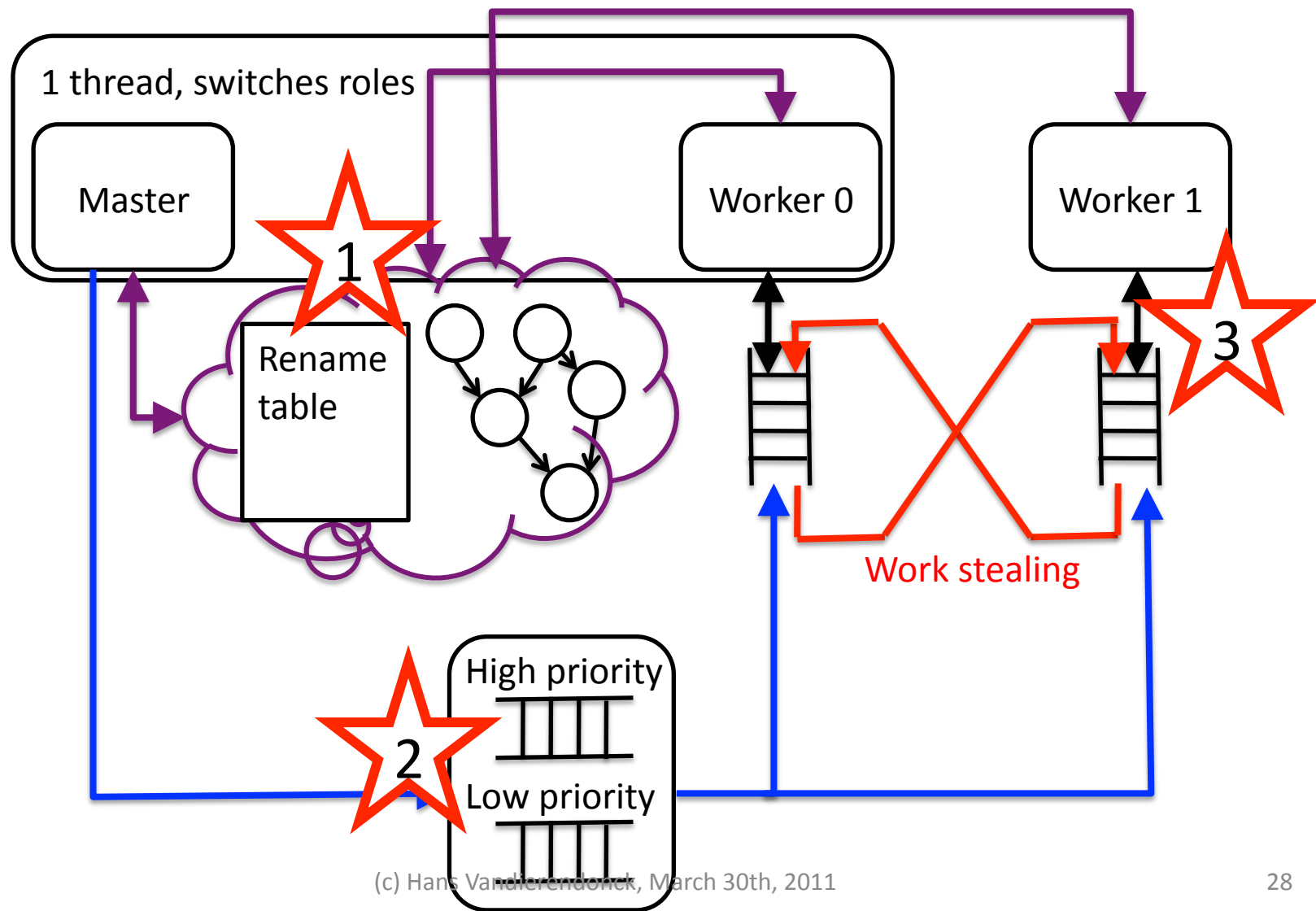


- Rename only if prior tasks read the same data but beware of the cost of copying!


Example: WAR With Renaming



SMPSS: When to Rename? When to Allocate Buffer?



SMPSS: Lazy Renaming

- **When to rename**
 1. When task is created
 2. When task is ready
 3. When task executes 
- **Advantage:**
 - Progressively higher probability that renaming will not be necessary!
 - Less overhead (allocation, copy-back)
 - Better for memory locality

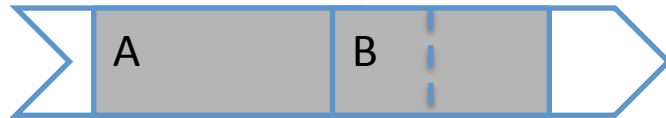
ALTERNATIVE APPROACHES TO MAINTAINING META-DATA

Memory Management

- **Tasks:**
 - Retrieving object metadata
 - Renaming of objects
 - Partially overlapping address ranges?
- **Main mechanisms**
 - SMPSS: hash table, indexed by “representant”
 - StarPU: handles
 - ADAM: block-based memory + meta-data allocator

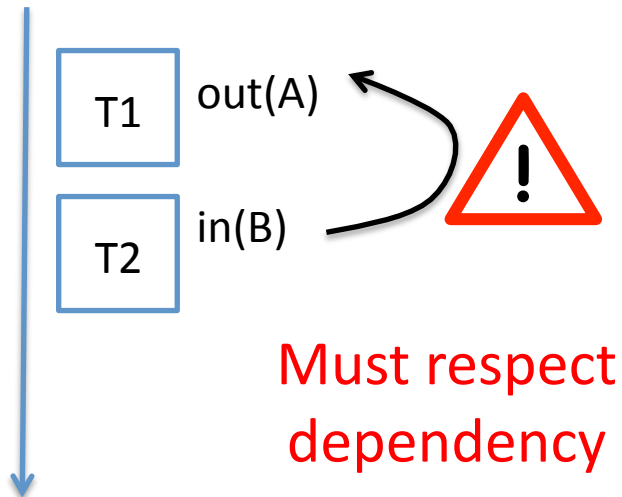
Partially Overlapping Address Ranges: General Case

Memory



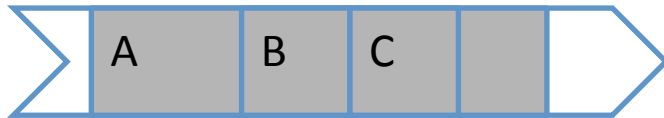
- Random address ranges
- Any address range currently in use may match!

Master thread



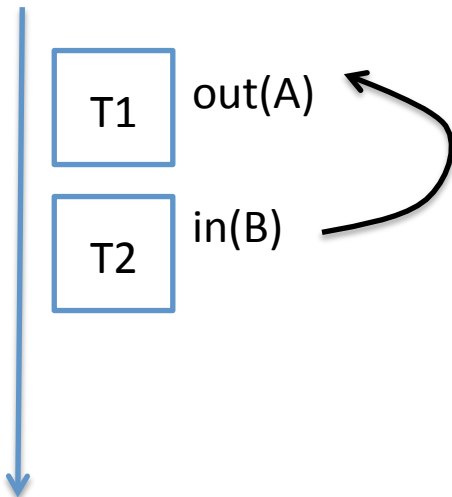
Overlapping Address Ranges: Ranges Contained in Other Ranges

Memory

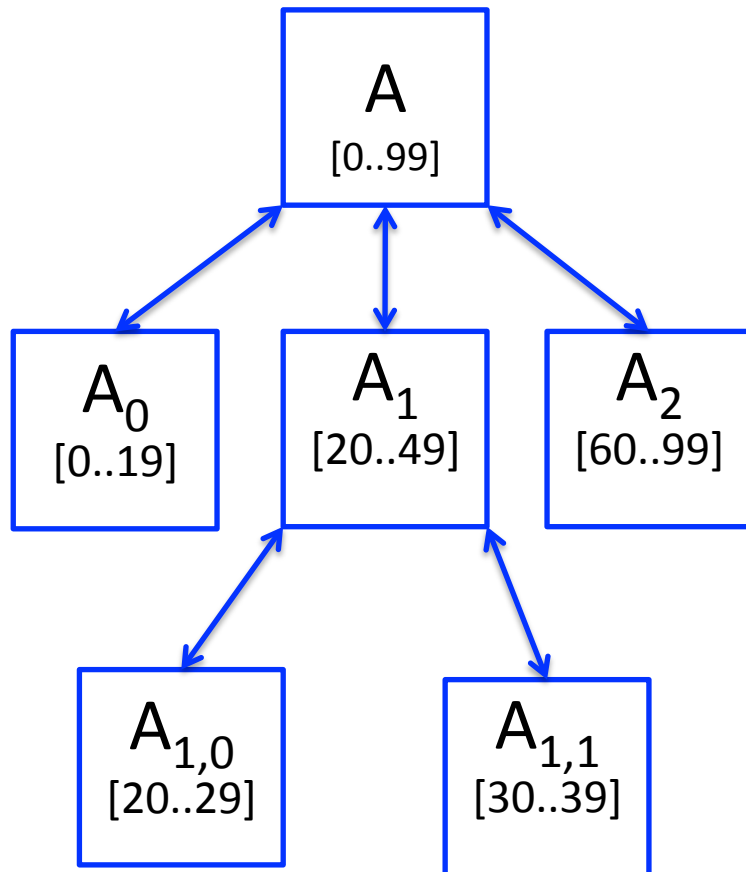


- Must know that B “is part of” A
- Tree of address ranges
- Hierarchical dependency checking

Master thread

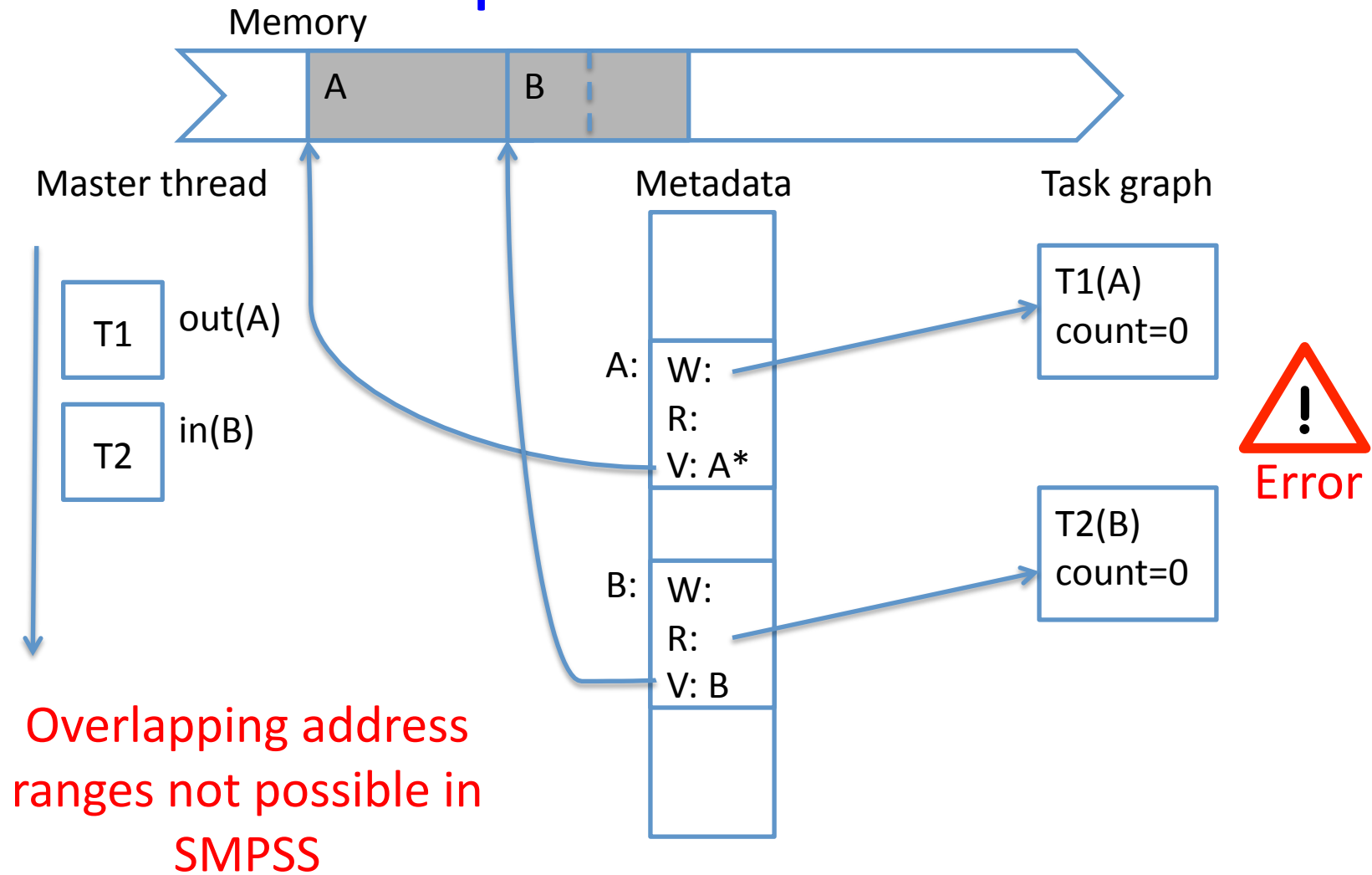


Address Ranges Tree



- Trace through tree when looking for tasks that read/write an address range
 - Ancestors
 - Descendants
- Example: $\text{out}(A_1)$ implies
 - $\text{out}(A_0)$
 - $\text{out}(A_{1,0}), \text{out}(A_{1,1})$
- Requires that user “registers” the address ranges tree with the runtime system

Overlapping Address Ranges vs Representants



StarPU Handles

- Handle stores object metadata
- Handle is passed as task argument, contains data address
- No hash table lookup required!

```
float vector[NX];

/* Declare data to StarPU runtime */
starpu_data_handle handle;
starpu_vector_data_register(&handle, 0, (uintptr_t)vector, NX, sizeof(vector[0]));

/* Create task */
struct starpu_task *task = starpu_task_create();
task->cl = &cl; /* Pointer to the codelet (= structure with a function pointer) */
task->buffers[0].handle = handle; /* First parameter of the codelet */
task->buffers[0].mode = STARPU_RW;

...
starpu_task_submit(task);
```

StarPU: Data Partitioning (1)

```
int vector[NX];

/* Declare data to StarPU */
starpu_data_handle handle;
starpu_vector_data_register(&handle, 0, (uintptr_t)vector, NX, sizeof(vector[0]));

/* Partition the vector in PARTS sub-vectors */
starpu_filter f = {
    .filter_func = starpu_block_filter_func_vector,
    .nchildren = PARTS,
    .get_nchildren = NULL,
    .get_child_ops = NULL
};
starpu_data_partition(handle, &f);

/* Get subdata number i (there is only 1 dimension) */
starpu_data_handle sub_handle = starpu_data_get_sub_data(handle, 1, i);
```

StarPU: Data Partitioning (2)

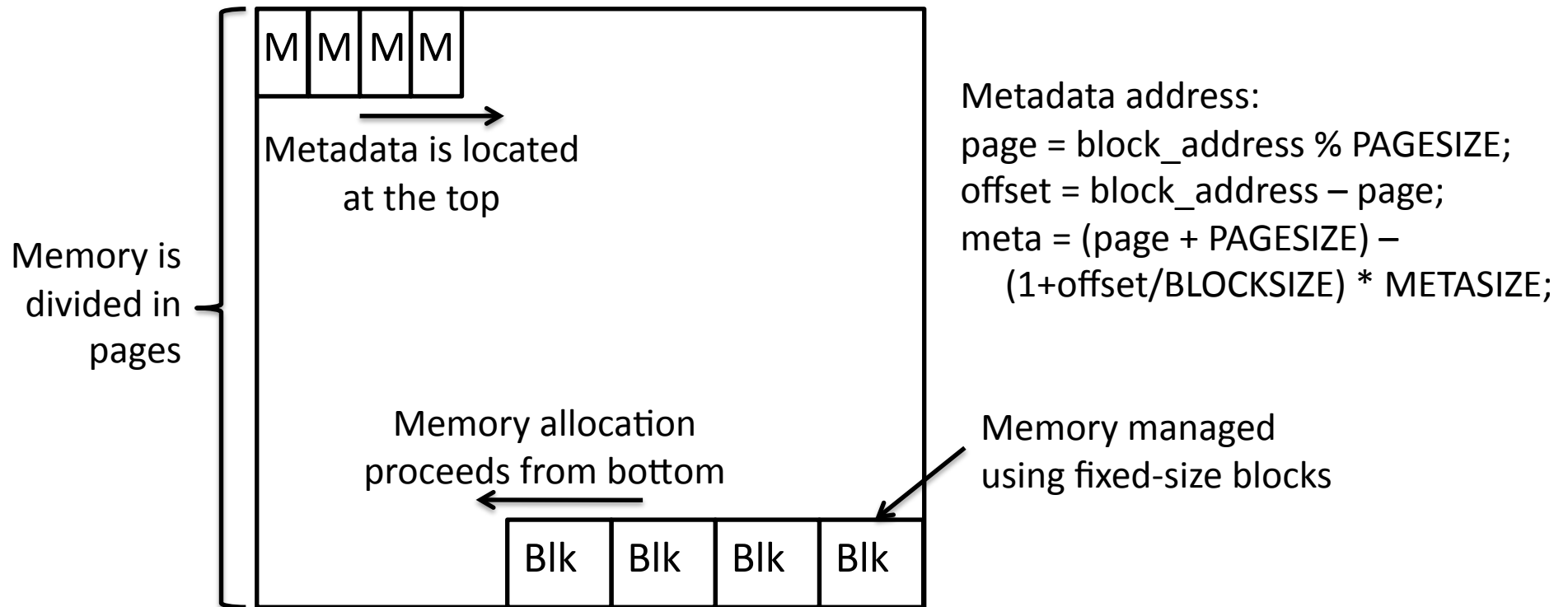
- **Benefits**
 - Data partitioning can be undone
 - Data partitioning can be redone differently,
 - Etc.
- **But: Dependency tracking is performed either on partitions or on the object itself**

ADAM: Integration with the Memory Allocator

- **Speedup data address to metadata mapping**
 - Replace memory allocation procedures
 - malloc(), calloc(), free(), ...
- **Handle overlapping memory ranges**
 - Divide address ranges in blocks
 - Programmer tunes granularity of blocks

ADAM

- Use arithmetic on the starting address to locate object metadata



ADAM

- **Block-based dependency tracking**

- Inuitive programming model

```
#pragma task inout(A[N])  
void taskone( float * A, unsigned N );
```

- Runtime system expands A[N] to an array of blocks

```
for( i=0; i < num; ++i ) {  
    if( (i%2) == 0 )  
        taskone( &A[i], 2 ); // Two blocks: A[0], A[1]; A[2], A[3]; ...  
    else  
        taskone( &A[i], 1 ); // One block: A[1]; A[3]; ...  
}
```

REDUCTIONS

SMPSS: Reductions

- What is a reduction?

- Inuitively:

```
float A[N];  
...  
float sum = 0;  
for( i=0; i < N; ++i )  
    sum += A[i];
```

- Summation can be parallelized, because of associativity

$$A[0]+...+A[N-1] = (A[0]+...+A[k-1]) + (A[k]+...+A[N-1])$$

- And because of commutativity

$$A[0]+...+A[N-1] = (A[k]+...+A[N-1]) + (A[0]+...+A[N-1])$$

- Neither property is exact on floating-point arguments!

SMPSS: Reductions

- **SMPSS requires associativity and commutativity**
 - Lock reduction variable to make update atomic
 - Necessary assumption: fraction of time spent in critical section is small compared to total task time
- **Scheduler modified to allow parallelism**
 - No dependency edges on inout between reducing tasks

```
#pragma css task input(A) inout(sum) reduction (sum)
void accum (float A[BS], float *sum) {
    int i, local_sum=0;
    for (i = 0; i < BS; i++)
        local_sum += A[i];
    #pragma css mutex lock (sum)
        *sum = *sum + local_sum;
    #pragma css mutex unlock (sum)
}
// Main code:
for (i=0; i<N; i+=BS)// sum(C[i])
    accum (&C[i], &sum);
```

(c) Hans Vandierendonck, March 30th, 2011

Not All Interesting Reductions are Commutative!

- **Cilk++ definition of reductions**
 - Algebraic monoid = tripple $\langle T, \times, e \rangle$
 - T a set of objects, e.g. int's
 - \times an associative operation
 - with identity e : for any x in T : $x \times e = x = e \times x$
- **Examples:**
 - Appending to a list
 - $T = \text{std::list}\langle \rangle$, $\times = \text{push_back}()$
 - Writing to an output file
 - $T = \text{character strings}$, $\times = \text{string concatenation}$

Reductions: Append to List

- **push_back()** in terms of list concatenation (++)
 - $l.\text{push_back}(x): \quad l \text{ ++ } (x)$
 - Identity: empty list ()
 $() \text{ ++ } l = l = l \text{ ++ } ()$
- **List concatenation is associative**
 - $(x \text{ ++ } y) \text{ ++ } z = x \text{ ++ } (y \text{ ++ } z)$
 - So it is a reduction operation in Cilk++
 - But not in SMPSS, Intel Threading Building Blocks,
...

***SS FOR ACCELERATORS: GPUSS**

Complete Memory Footprints

- Complete memory footprints describe all data touched by the task
- Advantages
 - Validating correctness (e.g. at compile-time)
 - Systems with non-shared memory
 - Cell BE (SPE local stores), GPU, clusters
 - Know what data to transfer and when to transfer



GPUSs: Language Extensions

```
#pragma css task
void matmul( float *A, float *B, float *C );

#pragma css targetdevice(cuda) implements(matmul)
void matmul_cuda( float *A, float *B, float *C ) {
    // tuned version for a CUDA-compatible device
}

#pragma css targetdevice(smp) implements(matmul)
void matmul_smp( float *A, float *B, float *C ) {
    // tuned vrsion for a SMP device
}
```

Organization of the Runtime

- Master thread generates task graph
 - Helper thread consumes tasks from task graph
 - Assigns tasks to GPUs
 - One worker thread/GPU
 - Perform data transfers
 - Invoke task on GPU
 - Retrieve results
 - GPUs have distinct main memories from CPU
 - Manage GPU memory as a software cache
 - Can be used to avoid data transfers by locality-aware scheduling
-  Deeply pipeline scheduler, ~ “double buffering” on Cell
- 

TICKET-BASED DEPENDENCY TRACKING

Overview

- Lazily maintain task graph
 - No explicit edges between tasks
- Organize task graph by depth of tasks
 - Walk over tasks at a particular depth when locating ready tasks
- Ticket-based dependency tracking
 - Efficient, lock-free
- How can this be efficient?
 - Extensive use of lock-free algorithms
 - Locate ready tasks “close” to finishing tasks

Ticket Locks

- Fair queuing of customers
- One global counter
- One next counter



	Enqueue	Ready?	Dequeue
actions	ticket := next++	ticket = global	++global

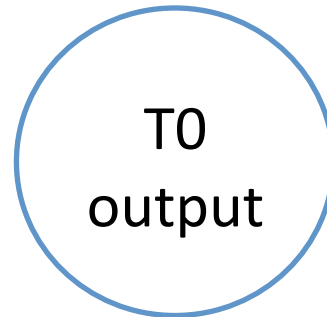
Ticket-Based Dependency Tracking

- Two queues: readers, writers
- Allow multiple tasks to wait on the same ticket

	Enqueue	Ready?	Dequeue
input	++R.next w := W.next	w = W.global	++R.global
output	if R.next != R.global then rename() ++W.next	true	++W.global
inout	r := R.next++ w := W.next++	r = R.global and w = W.global	++R.global ++W.global

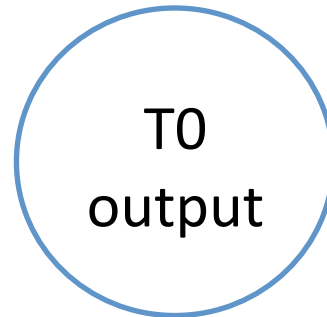
Ticket-Based Dependency Tracking

R.next = 0
R.global = 0
W.next = 0
W.global = 0



Ticket-Based Dependency Tracking

R.next = 0
R.global = 0
W.next = 1
W.global = 0



Now assume that T0 keeps
executing for some time

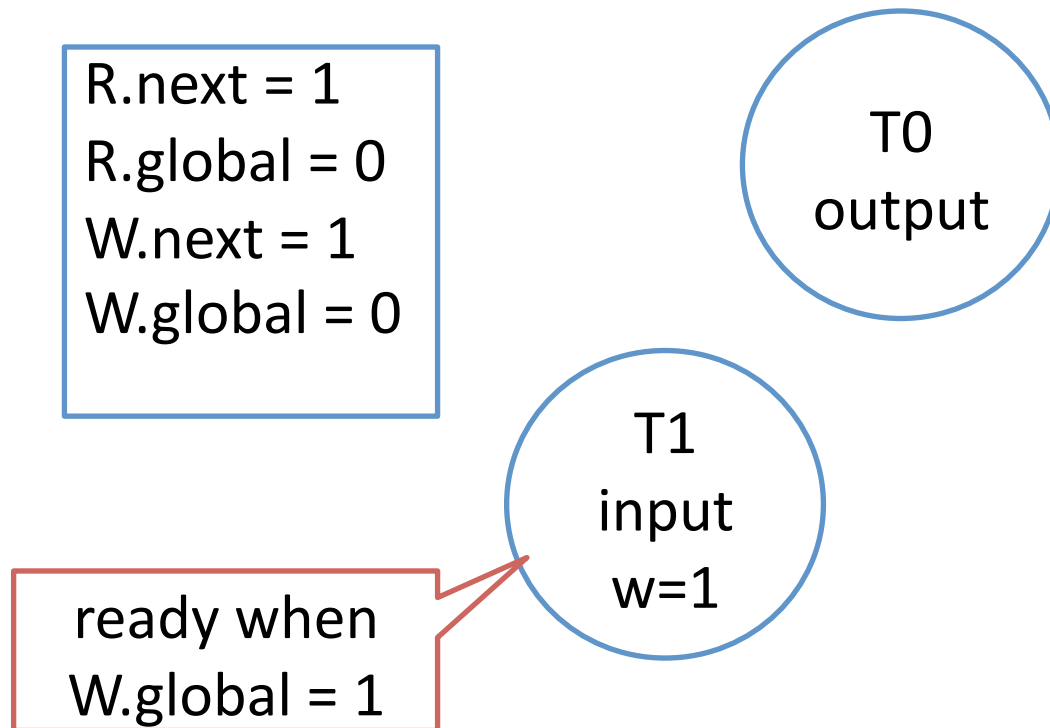
Ticket-Based Dependency Tracking

R.next = 0
R.global = 0
W.next = 1
W.global = 0

T0
output

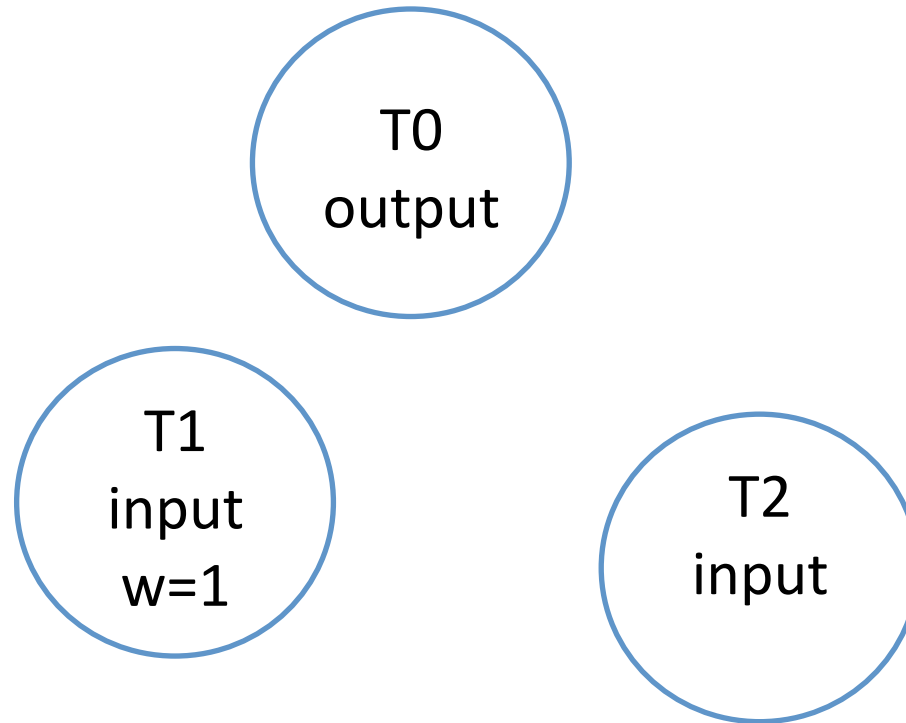
T1
input

Ticket-Based Dependency Tracking

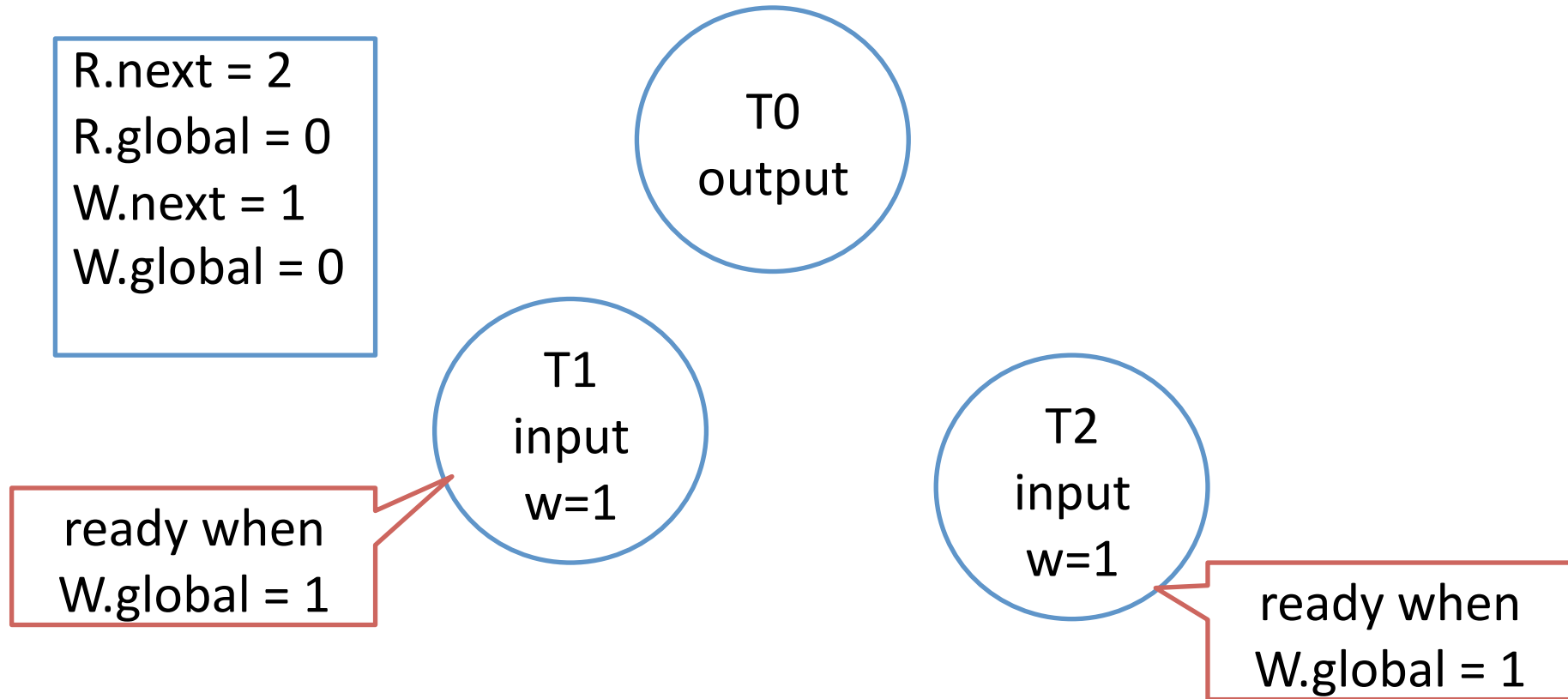


Ticket-Based Dependency Tracking

R.next = 1
R.global = 0
W.next = 1
W.global = 0

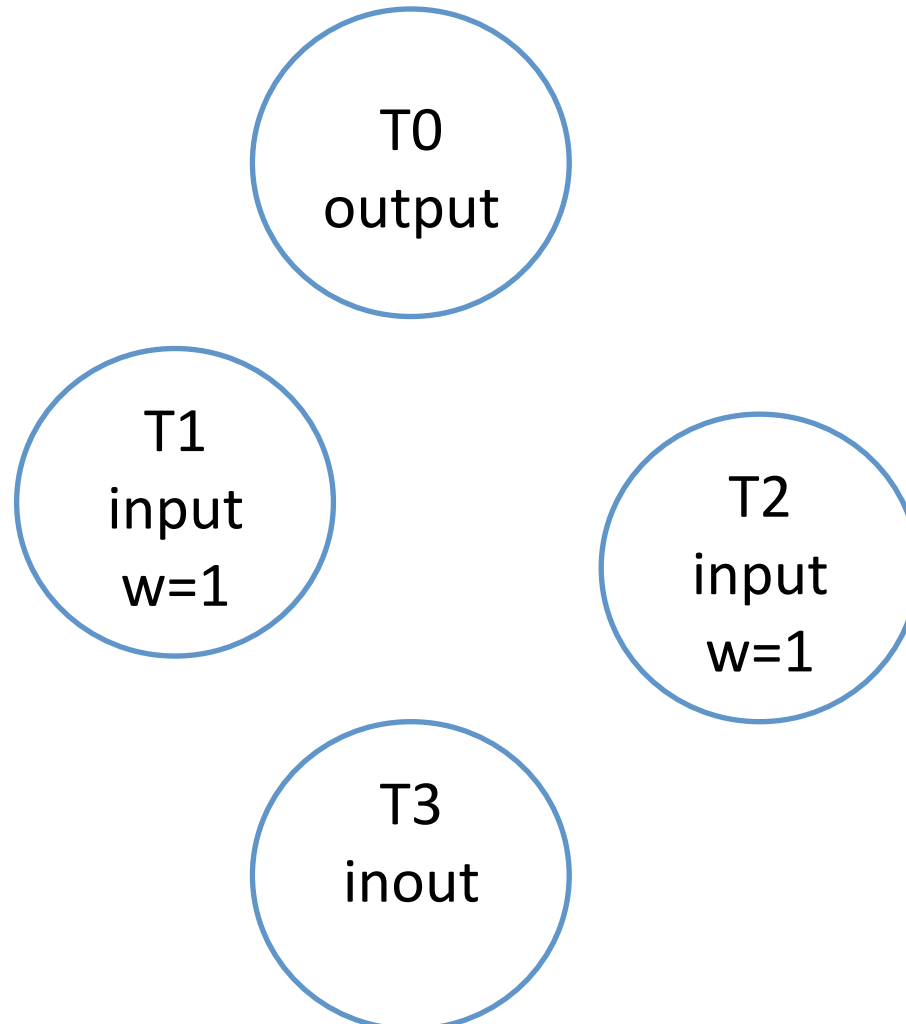


Ticket-Based Dependency Tracking

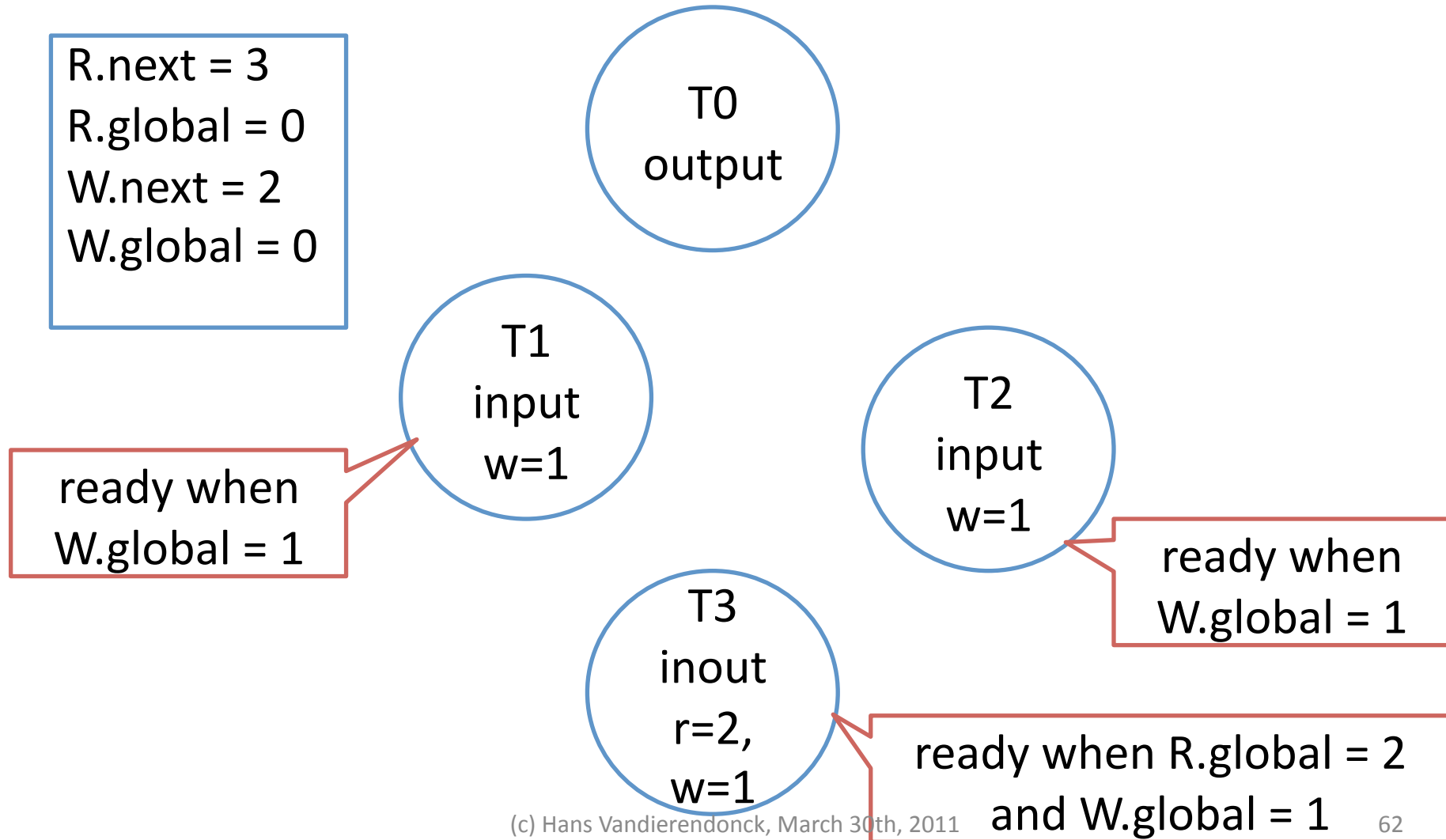


Ticket-Based Dependency Tracking

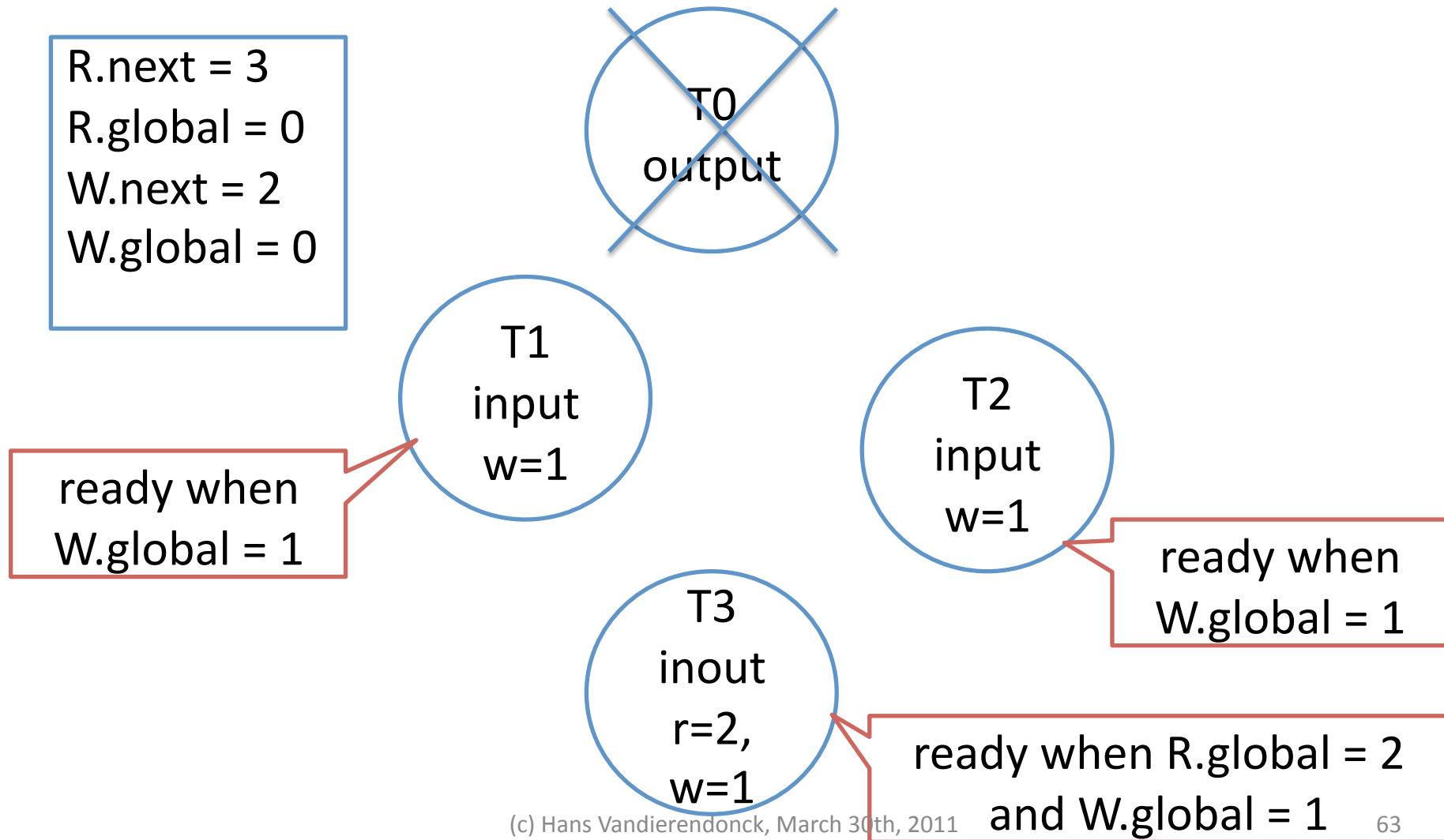
R.next = 2
R.global = 0
W.next = 1
W.global = 0



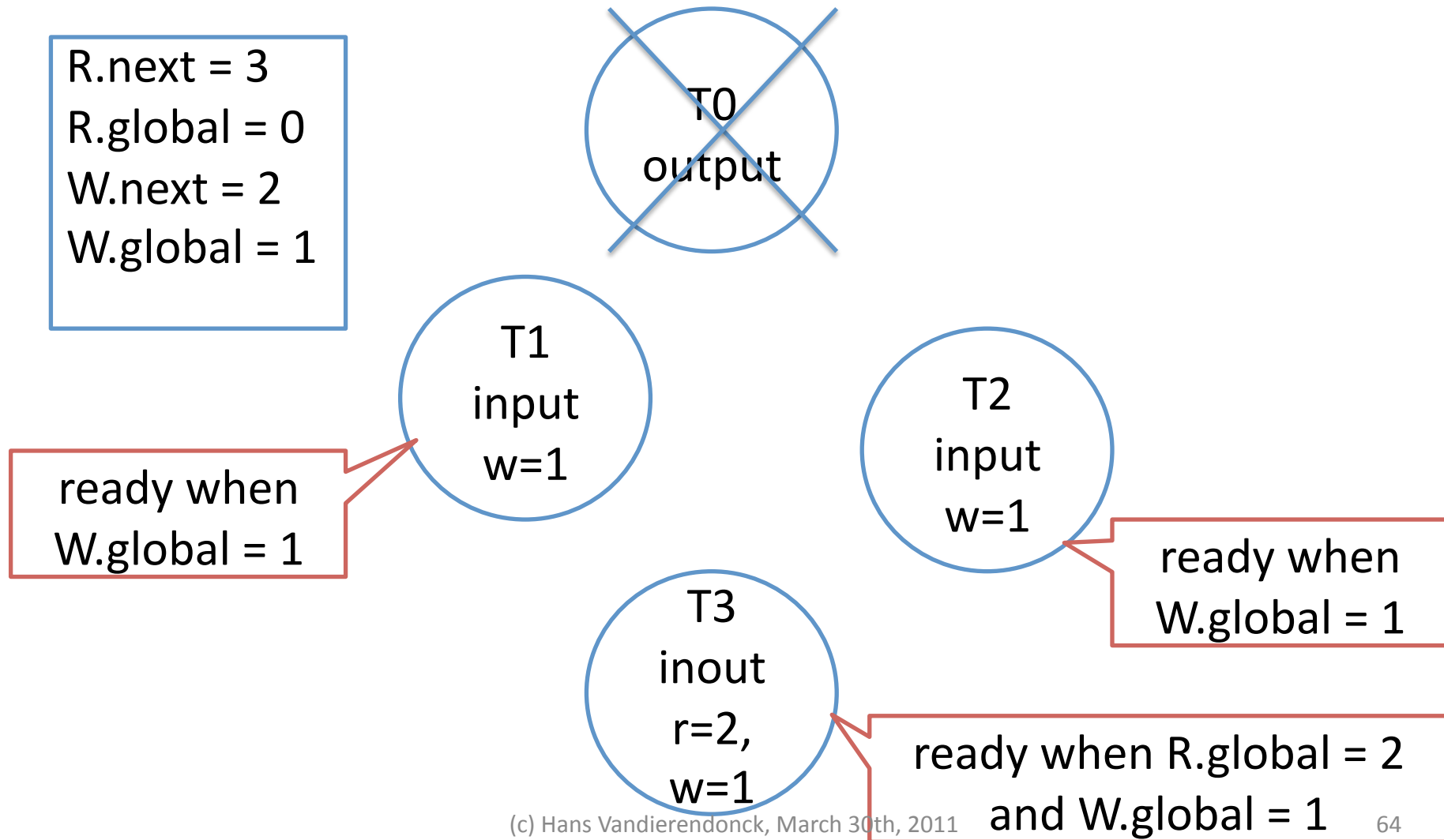
Ticket-Based Dependency Tracking



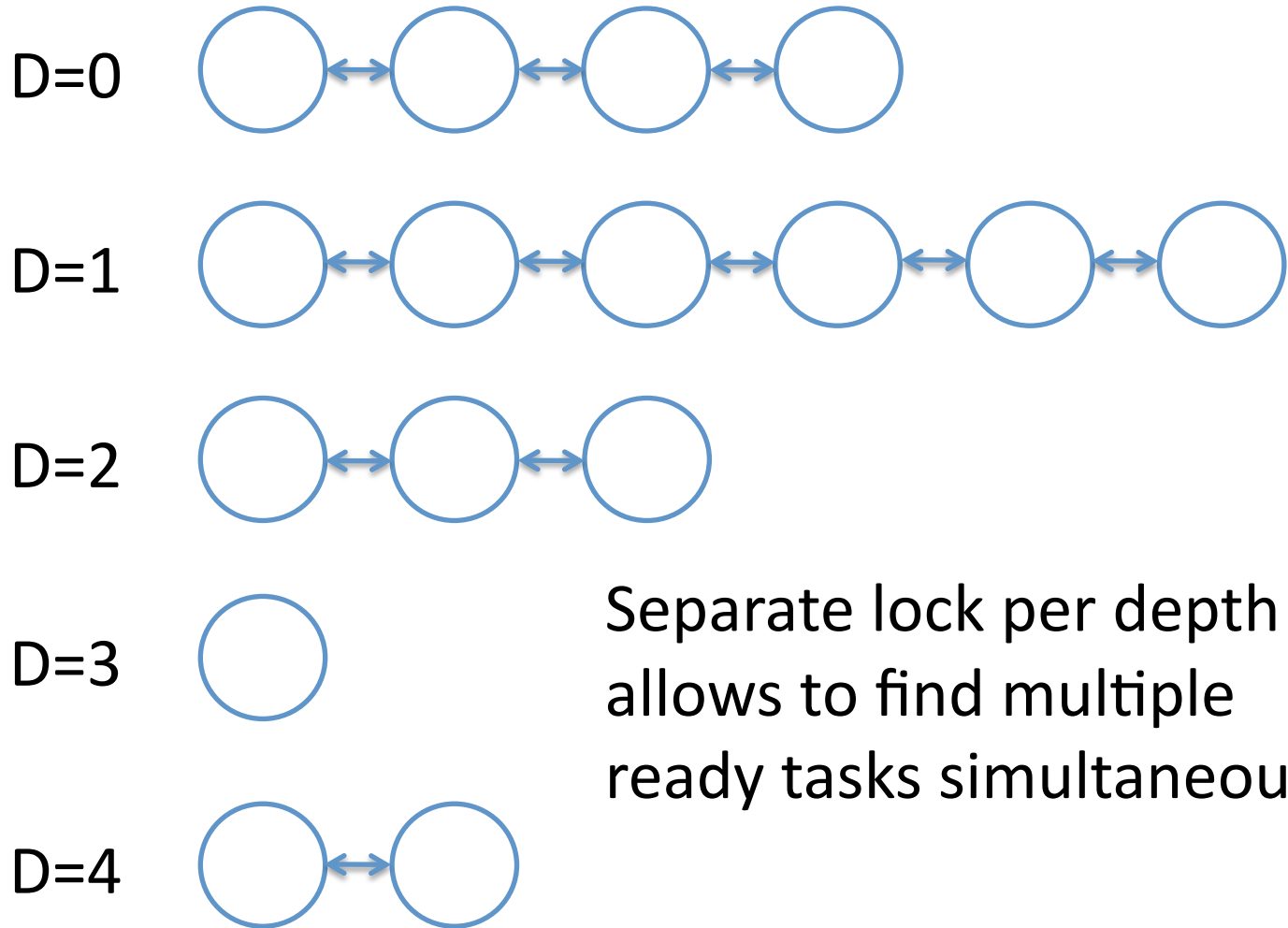
Ticket-Based Dependency Tracking



Ticket-Based Dependency Tracking

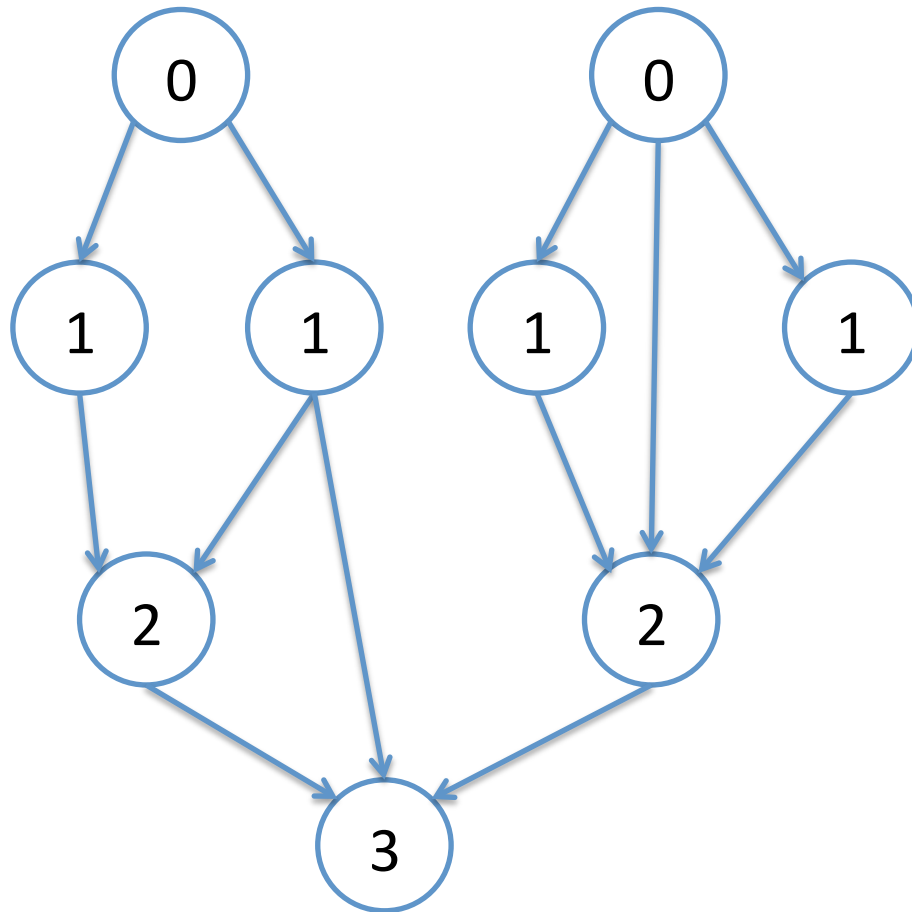


Task Graph Organization



Separate lock per depth
allows to find multiple
ready tasks simultaneously

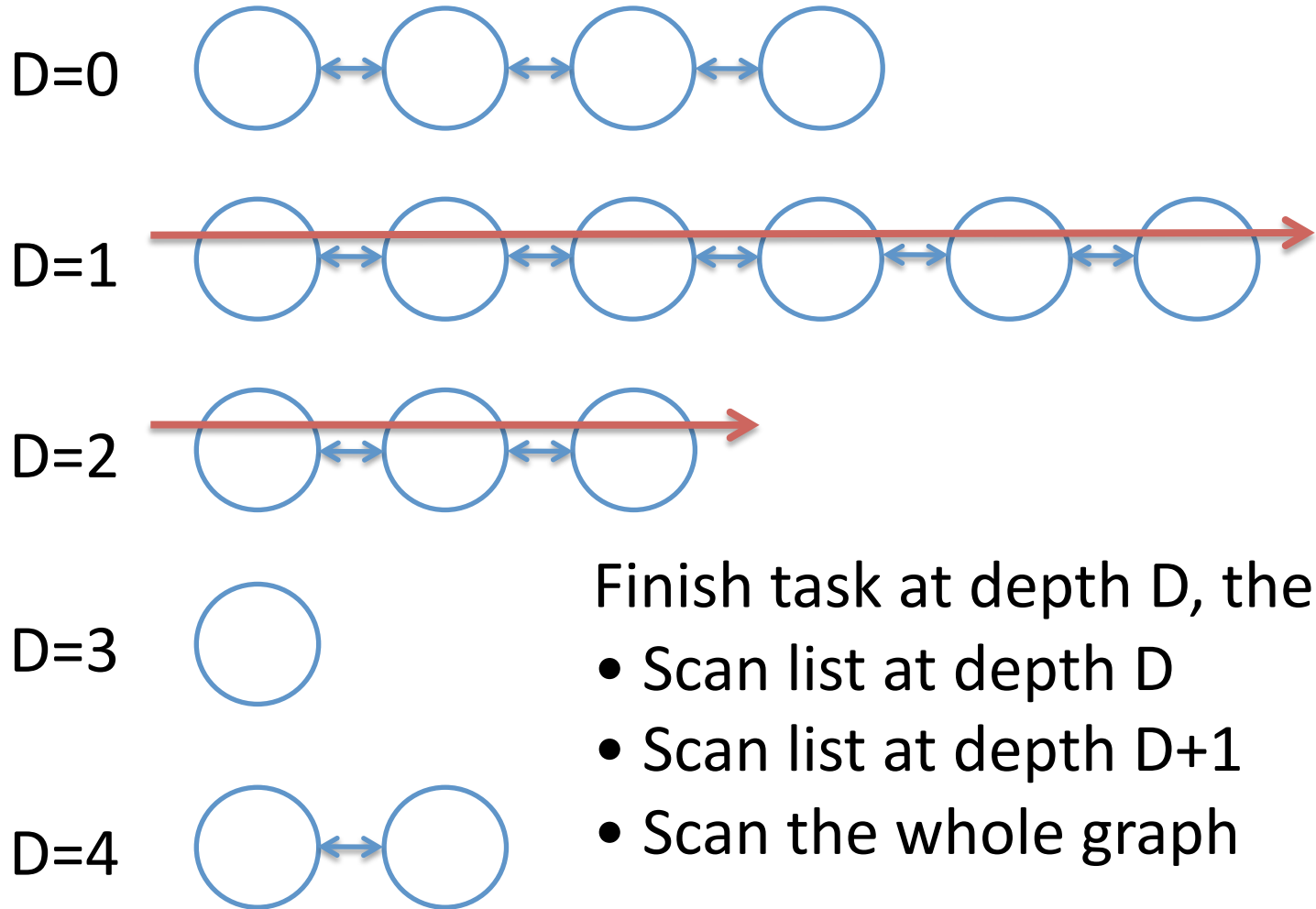
Task Depth



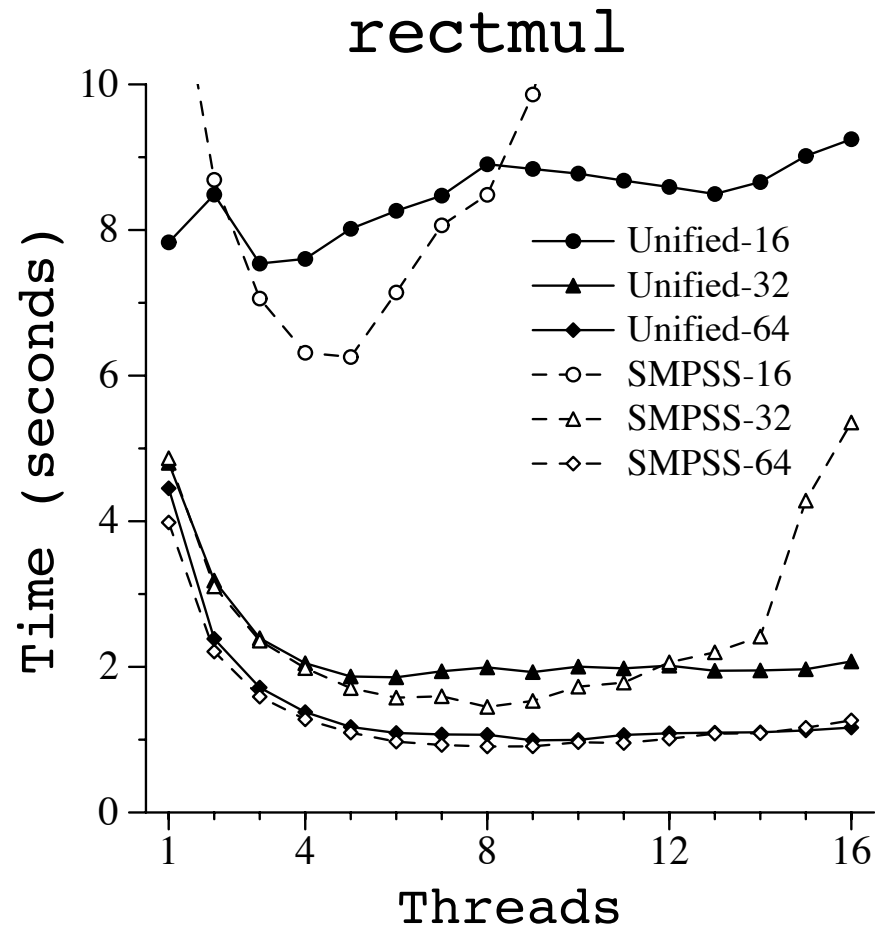
Earliest execution time,
assuming unit execution
times

Critical path length:
highest execution time

Task Graph Organization

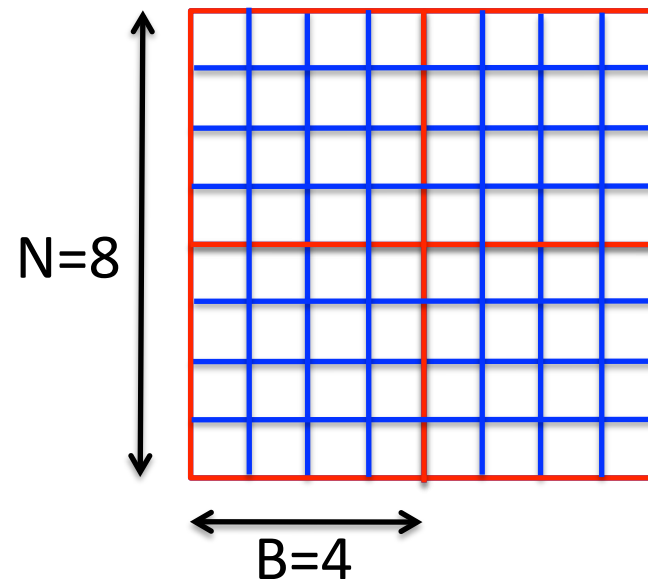


Implemented in a Unified Cilk+Task Dependency-Aware Scheduler



Background: The Hypermatrix

- The hypermatrix is a storage format for blocked matrices
- A $N \times N$ matrix is blocked in D blocks of size B such that $N = DB$
- Multiple levels of pointers are possible
- This storage format is generally good for caches (locality)



Background: The Hypermatrix

- The hypermatrix is a storage format for blocked matrices
- A $N \times N$ matrix is blocked in D blocks of size B such that $N = DB$
- Multiple levels of pointers are possible
- This storage format is generally good for caches (locality)

