Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Developing HPC Applications with Task-Aware Libraries

Kevin Sala and Xavier Teruel

21/06/2024

HEART 2024, Porto

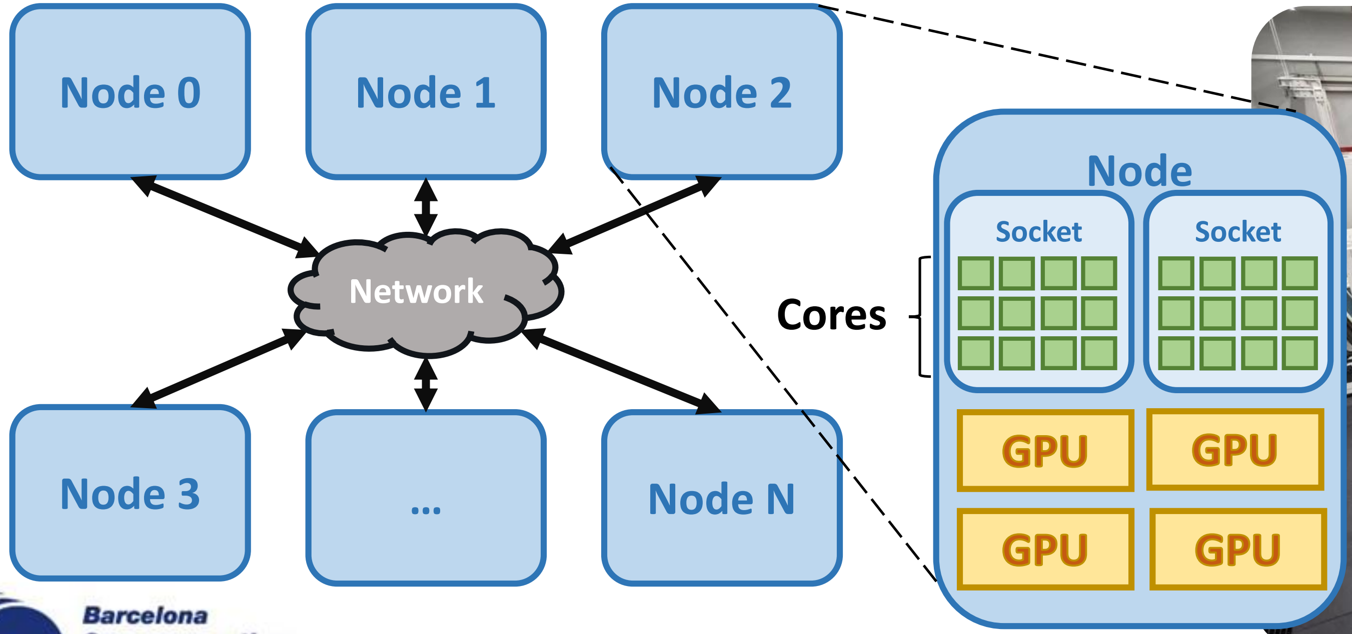
Outline

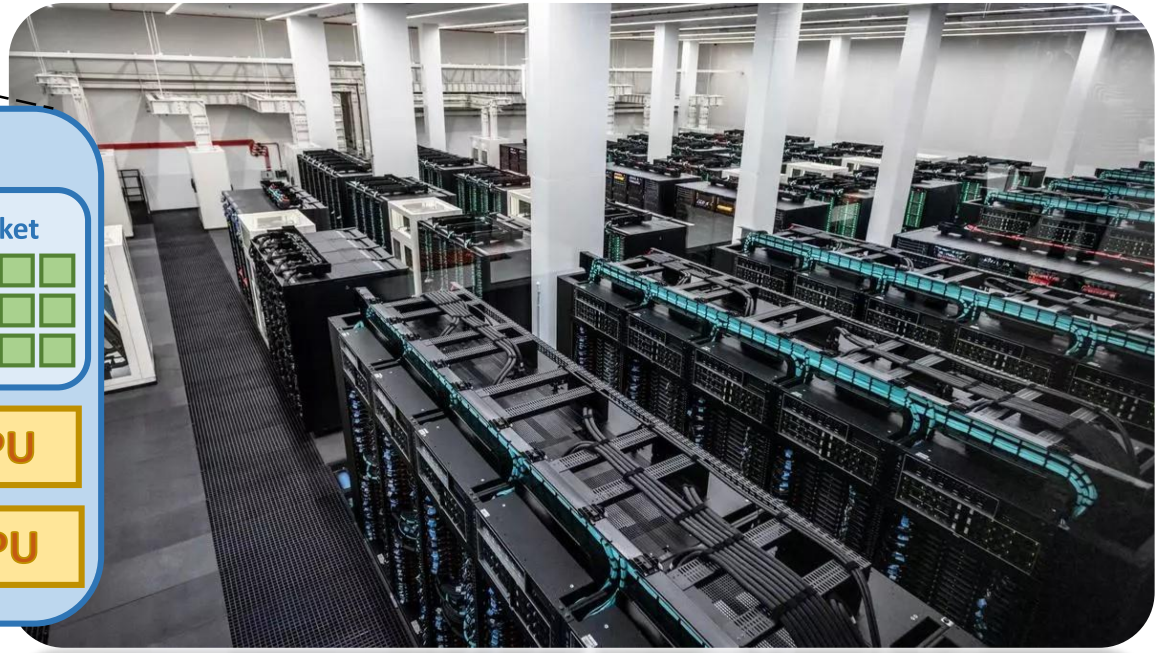
- **Motivation**
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

HPC Systems Keep Growing

Hundreds of thousands of **computing nodes**

- Already **hundreds of cores** per node
- Most with multiple GPU devices per node
- Connected by high-end **network interconnects**





Marenostrum5 @ BSC. Photo: Àngel García

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Many Components and APIs

Large **multi-core** processors

- x86, ARM, RISC-V

Discrete **accelerators**

- GPUs, FPGAs and ML

High-performance **memory** & **storage** systems

- SSDs and NVM

High-performance **networks**

- Smart NICs

Many APIs!

Linux io_uring

Intel PMDK

NVIDIA DOCA

Programming HPC applications

Developing HPC applications is becoming increasingly difficult!

- HPC applications must combine some of those APIs
  - **Inter-node** communication: **MPI**
  - **Intra-node** parallelism: **OpenMP**, OmpSs-2
  - **GPU** parallelism: **CUDA**, HIP, SYCL, etc.
  - **I/O** operations

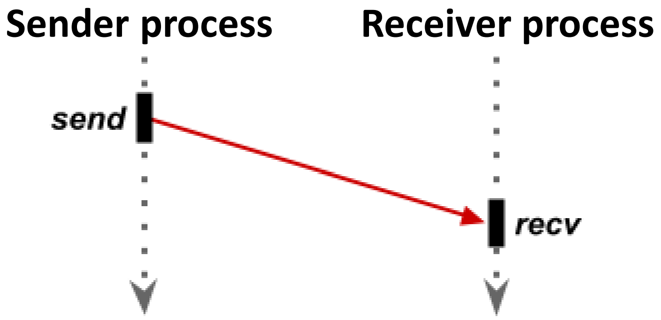
- To achieve high performance → **non-blocking** and **low-level** operations

- **Efficiently combining APIs** is not that easy...

Programming HPC applications

MPI - **Message Passing Interface**

- **Multiple OS processes** execute the application
- Different virtual memory spaces
- Processes explicitly communicate with **data messages**
  - MPI_Send, MPI_Recv (**blocking** operations)
  - MPI_Isend, MPI_Irecv (**non-blocking** operations)
  - MPI_Bcast, MPI_Barrier (collective operations)

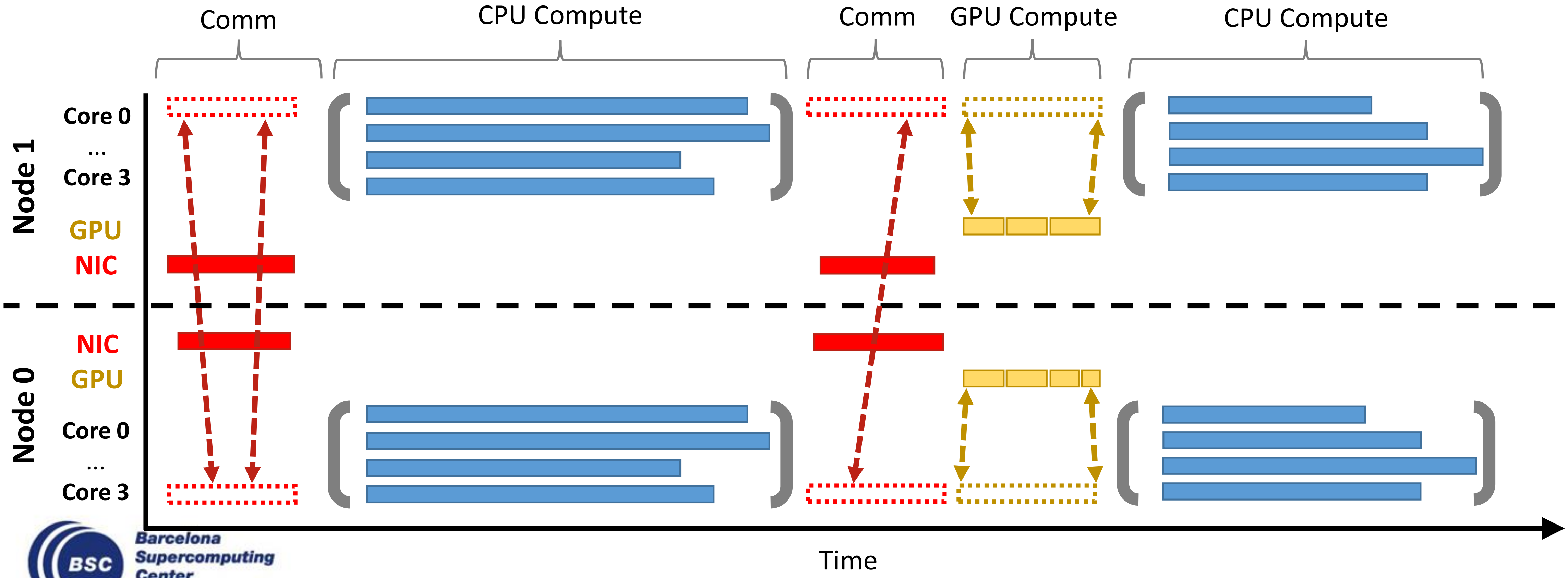


CUDA - Offloading Computation Kernels to the GPUs

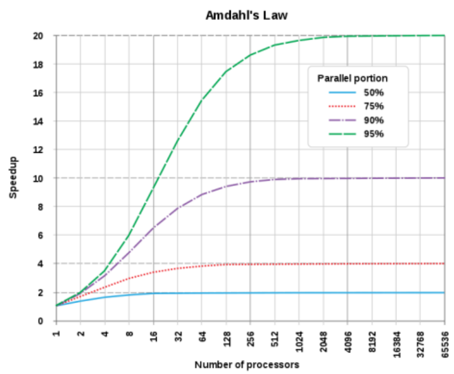
- Host (CPUs) offloads kernels to the GPU
- Memory copies between the host and GPU

Hybrid Applications

Fork-Join model (aka Bulk Synchronous Parallelism) is the traditional

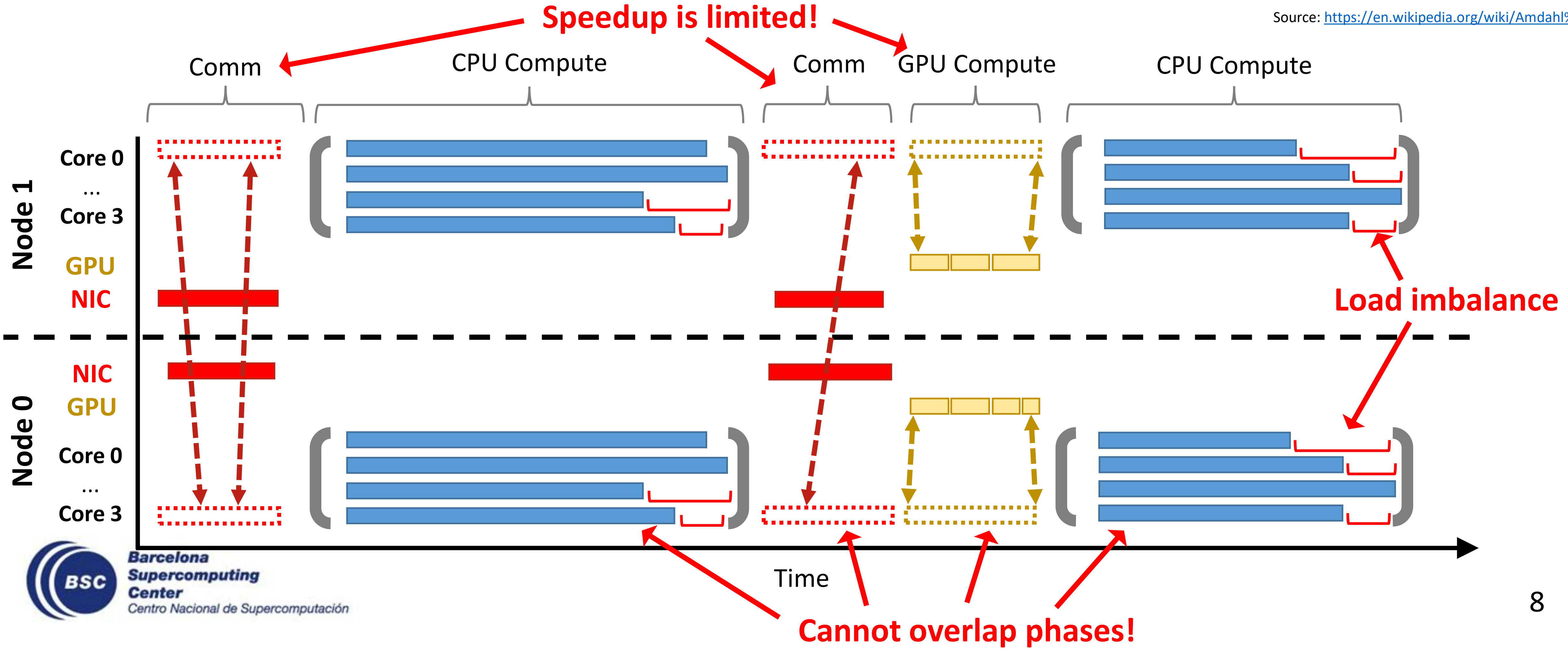


Hybrid Applications



Source: https://en.wikipedia.org/wiki/Amdahl%27s_law

Fork-Join model (aka Bulk Synchronous Parallelism) is the traditional

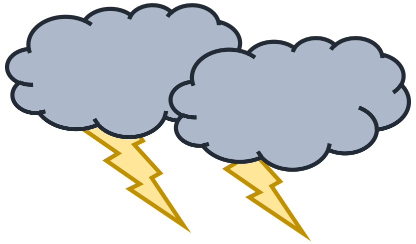


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hybrid Applications (II)

We could **parallelize communications** and GPU **offloading**

- Implement **overlapping** of phases on the application side
- Many **application modifications!**

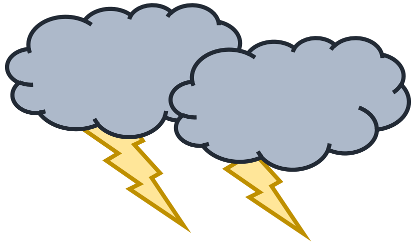


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hybrid Applications (II)

We could **parallelize communications** and GPU **offloading**

- Implement **overlapping** of phases on the application side
- Many **application modifications!**



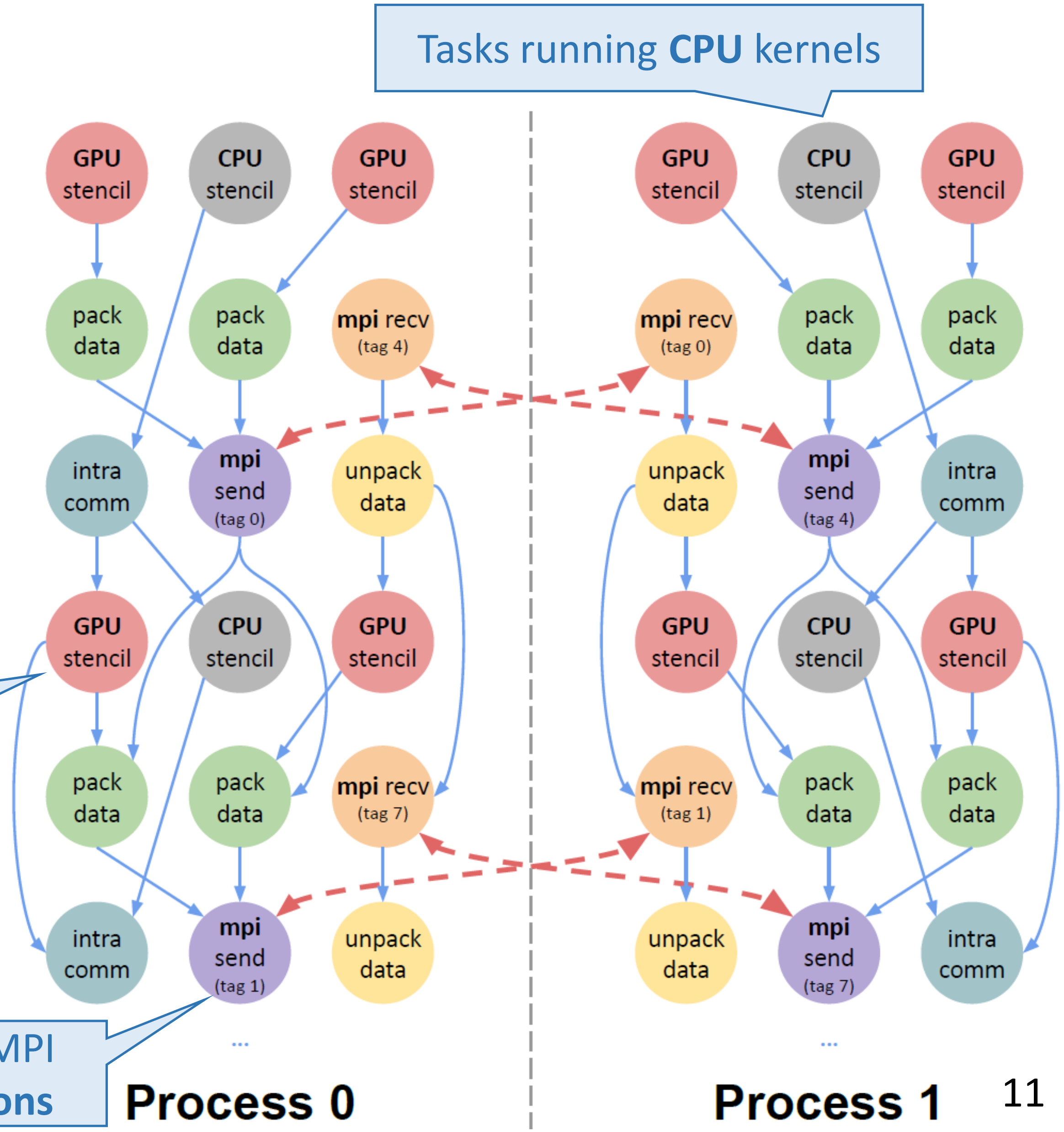
Can **tasking models** help us **orchestrating** all this parallelism and interactions?

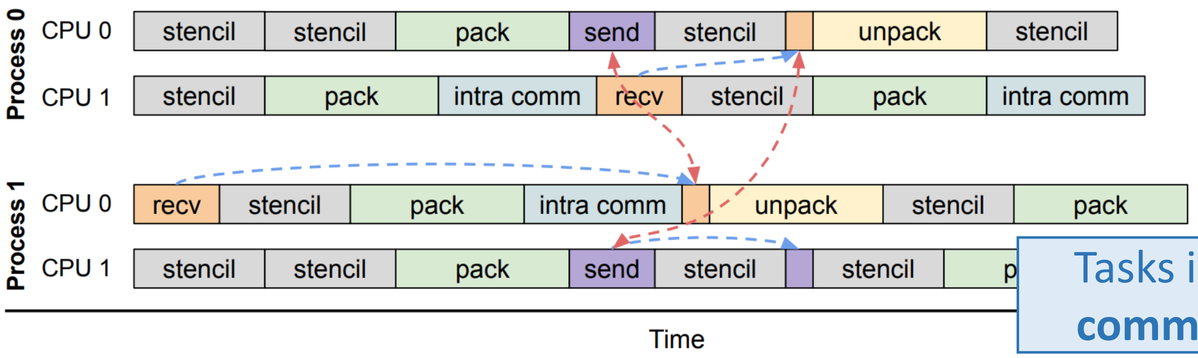
Data-Flow Model with Tasks + Dependencies

We can represent applications through task graphs

- Use **tasks** to express
  - **Computations**
  - **MPI communications** (message passing)
  - **GPU kernels** (offloading of kernels)
- Use **data dependencies** as the general **synchronization** mechanism

One **task graph per process**!





Data-Flow Model with Tasks + Dependencies

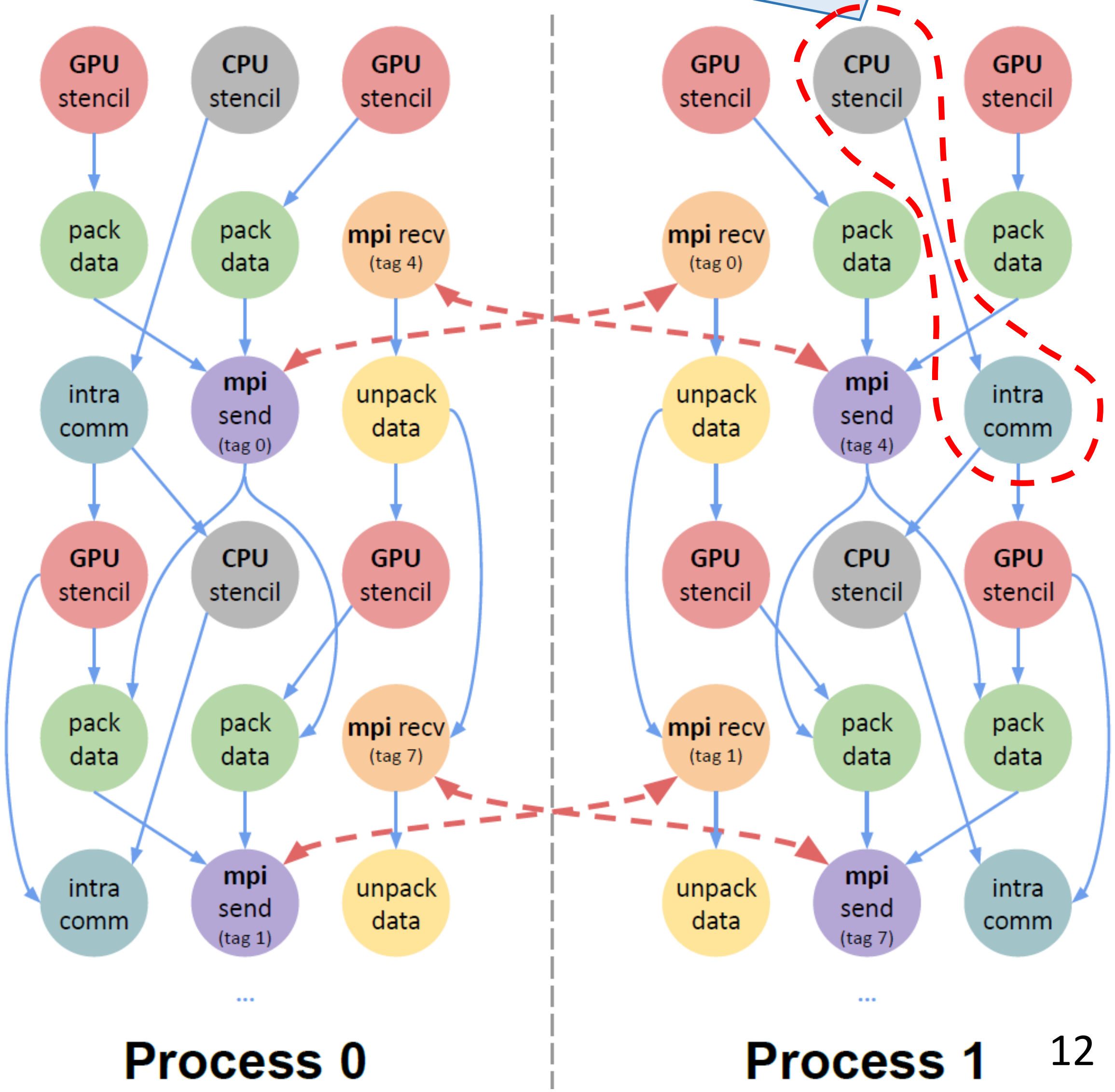
We can represent applications through task graphs

- Use **tasks** to express
  - **Computations**
  - MPI communications (message passing)
  - GPU kernels (offloading of kernels)

```
int block_i = ...;
int block_j = ...;

#pragma omp task depend(inout: data[block_i])
{
  stencil_cpu(&data[block_i]);
}

#pragma omp task depend(in: data[block_i]) depend(out: data[block_j])
{
  intracomm_copy(&data[block_i], &data[block_j]);
}
```

Regular memory copies too!

Tasks running **CPU** kernels



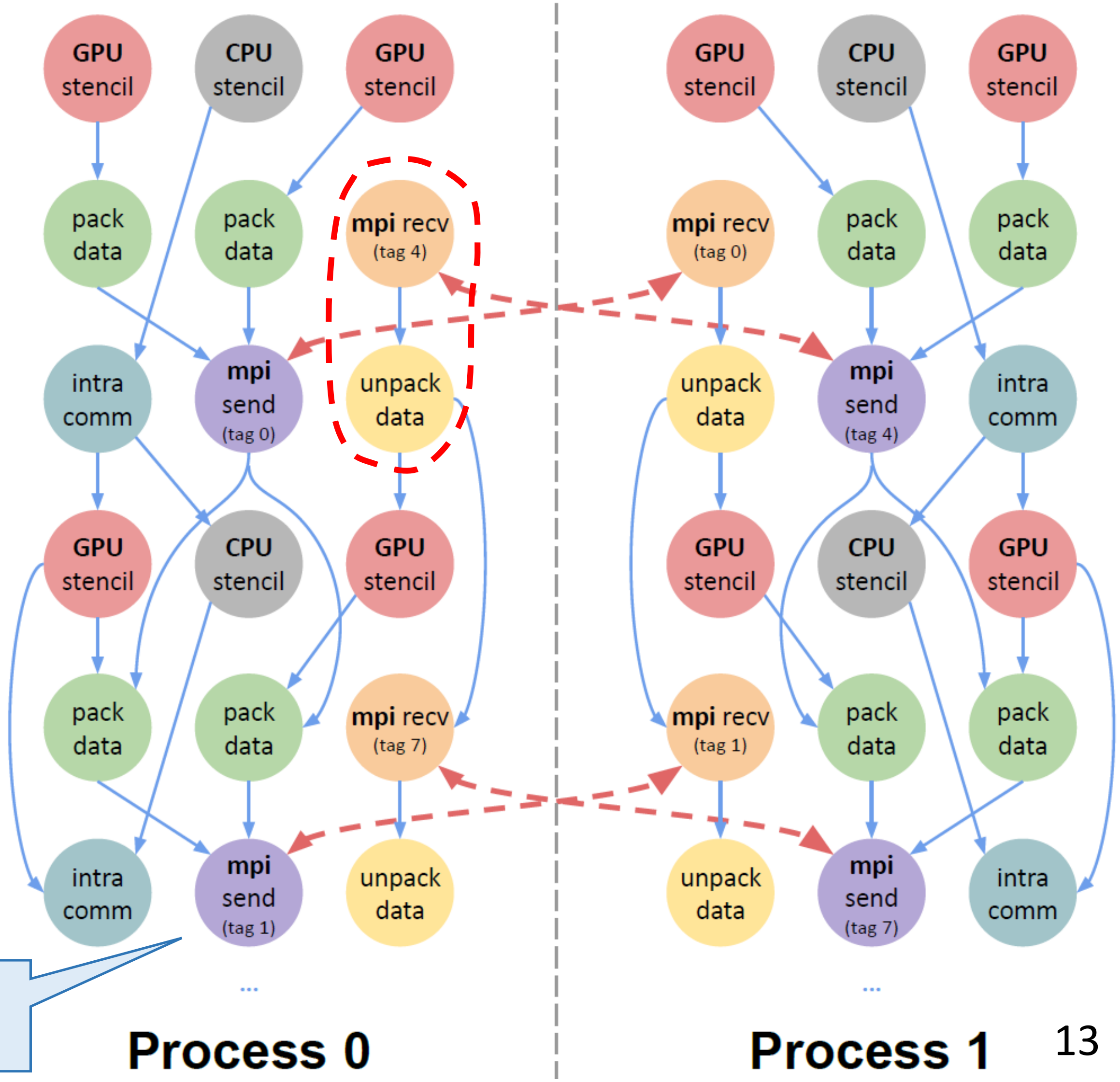
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Data-Flow Model with Tasks + Dependencies

We can represent applications through task graphs

- Use **tasks** to express
  - Computations
  - **MPI communications** (message passing)
  - GPU kernels (offloading of kernels)

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Recv(recvbuf[msg], msg_size, MPI_BYTE, src, tag, ...);
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```



Tasks issuing MPI **communications**

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

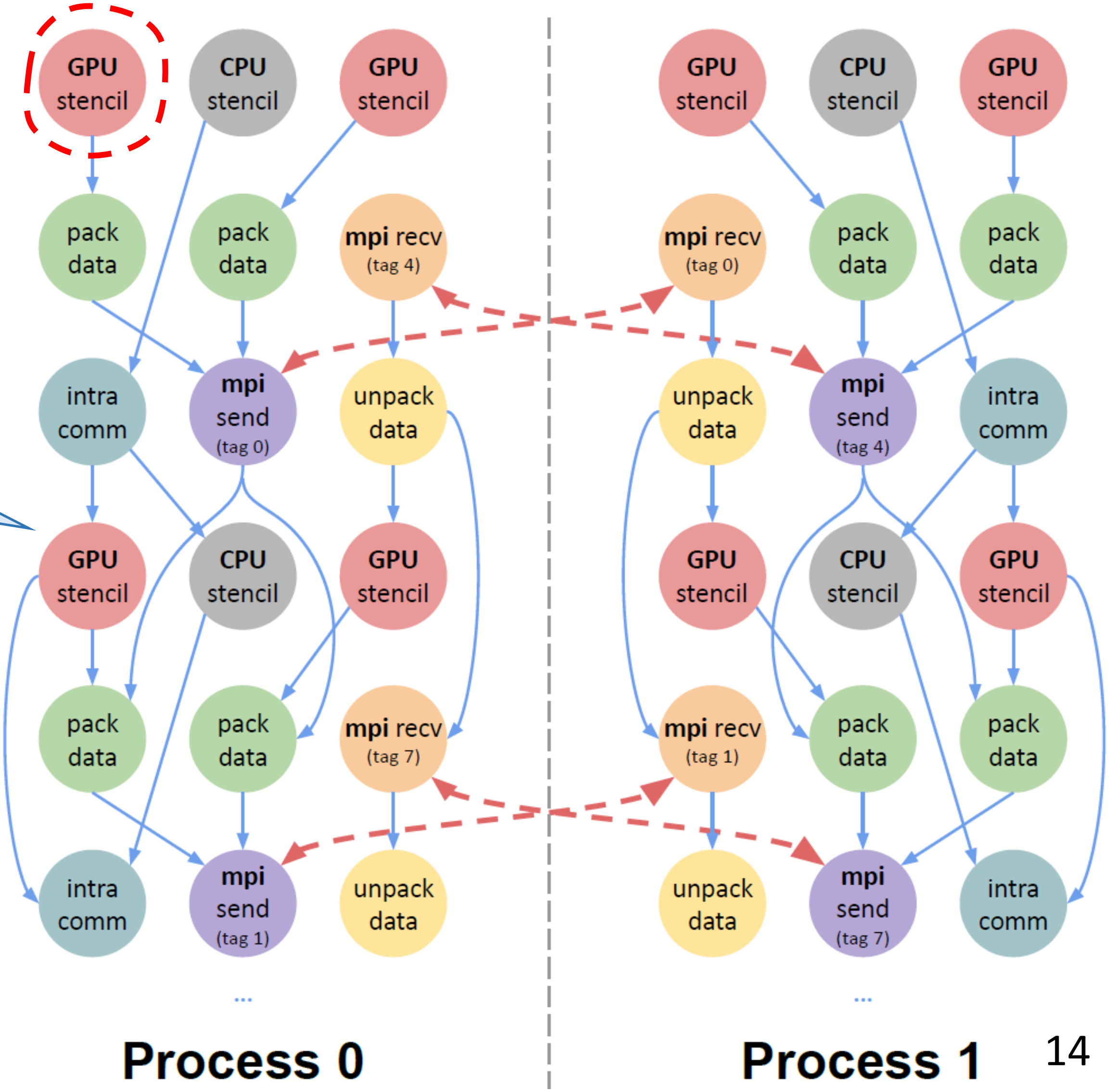
Data-Flow Model with Tasks + Dependencies

We can represent applications through task graphs

- Use **tasks** to express
  - Computations
  - MPI communications (message passing)
  - **GPU kernels** (offloading of kernels)

Tasks offloading to **GPU**

```
#pragma omp task depend(inout: data[block_i], d_data[block_i])
{
  cudaMemcpyAsync(d_data[block_i], data[block_i], size,
                  cudaMemcpyHostToDevice, stream);
  stencil_gpu<<<..., stream>>>(d_data[block_i]);
  cudaMemcpyAsync(data[block_i], d_data[block_i], size,
                  cudaMemcpyDeviceToHost, stream);
  cudaStreamSynchronize(stream);
}
```

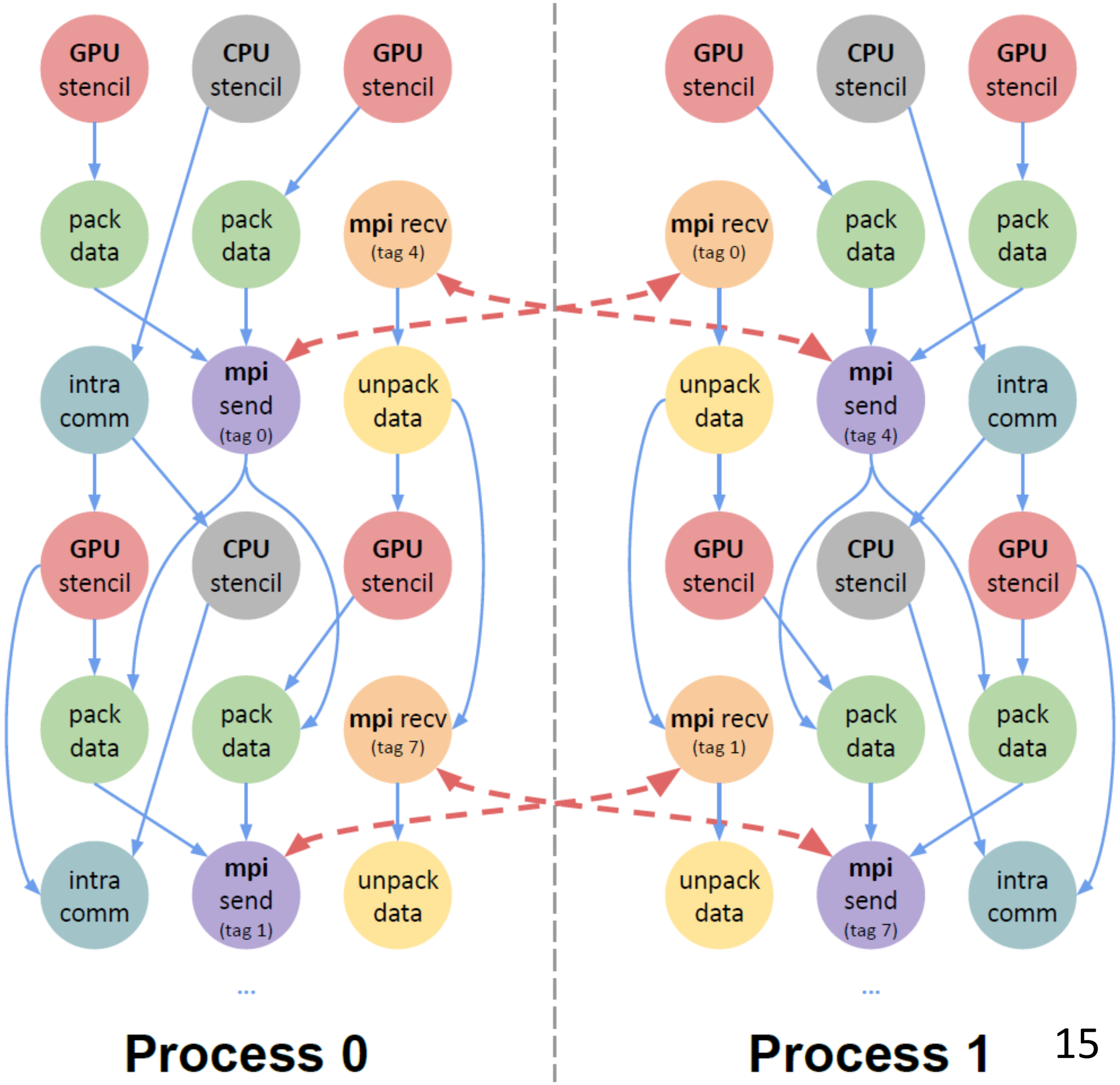


Data-Flow Model with Tasks + Dependencies

We can represent applications through task graphs

- Use **tasks** to express
  - **Computations**
  - **MPI communications** (message passing)
  - **GPU kernels** (offloading of kernels)

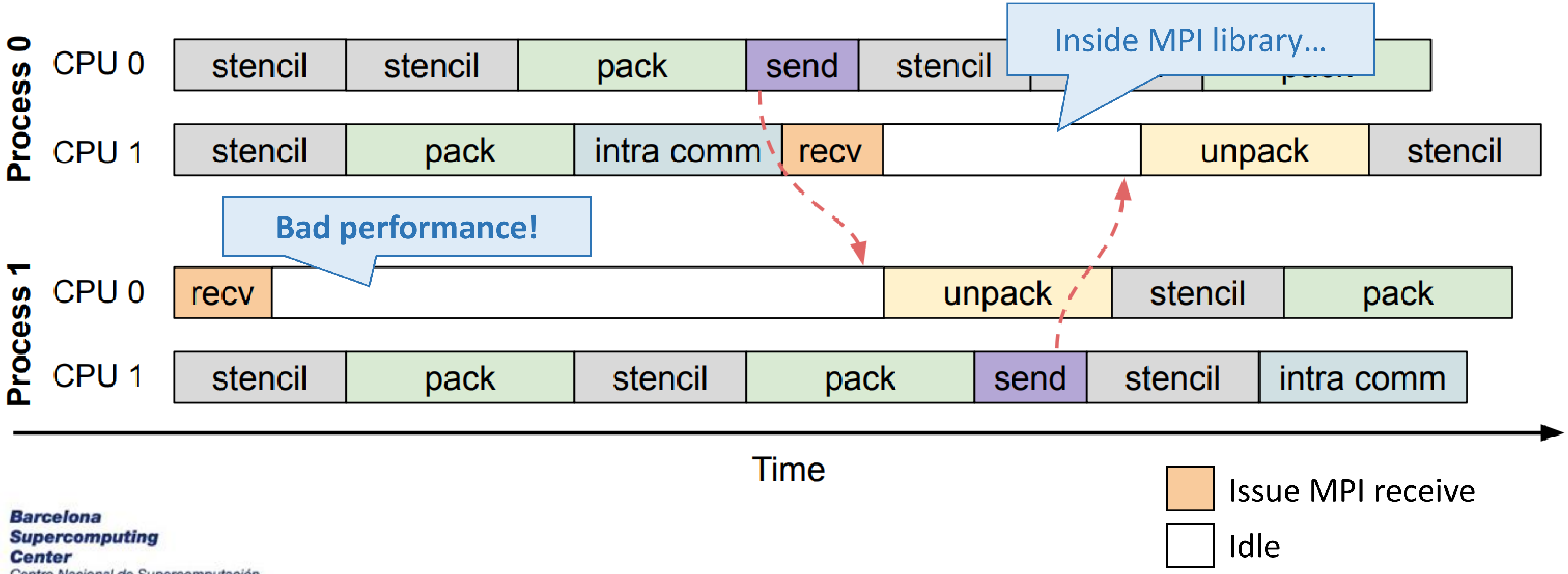
But there are several problems!



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

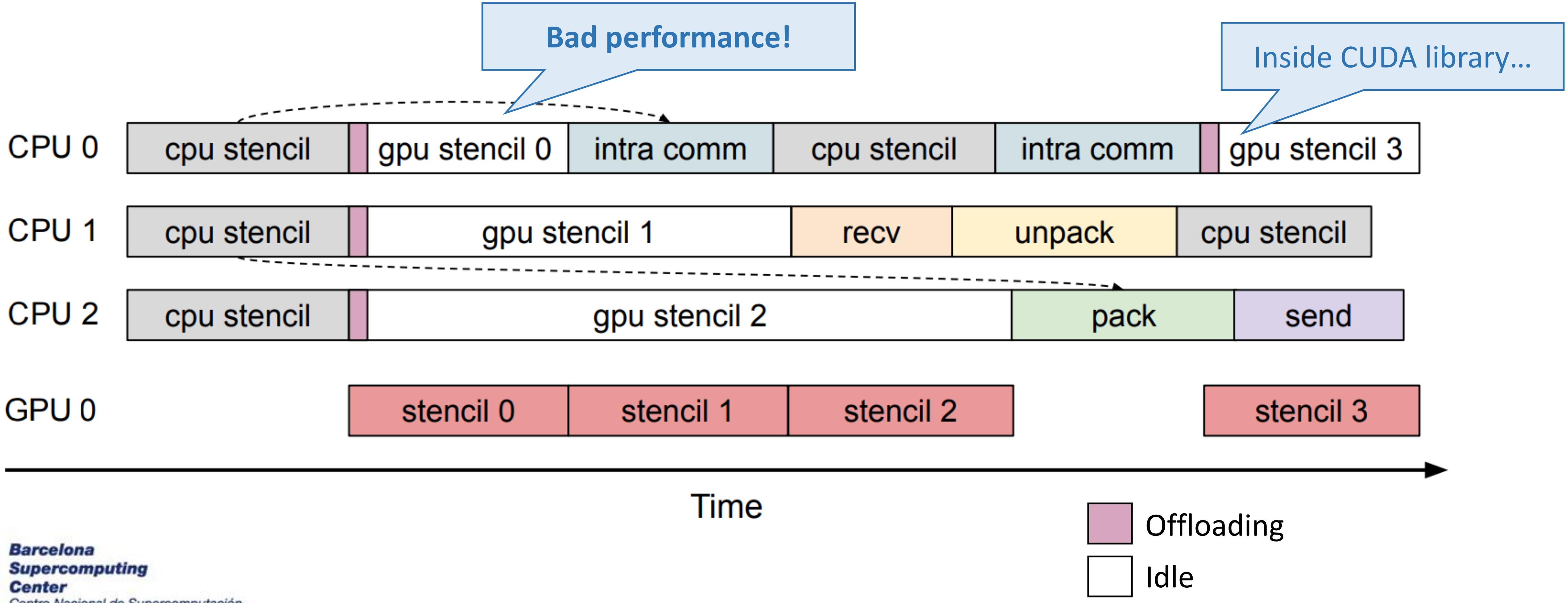
Challenges with Tasks + Operations

The application will **waste CPU resources**!



Challenges with Tasks + Operations (II)

The application will **waste CPU** resources!

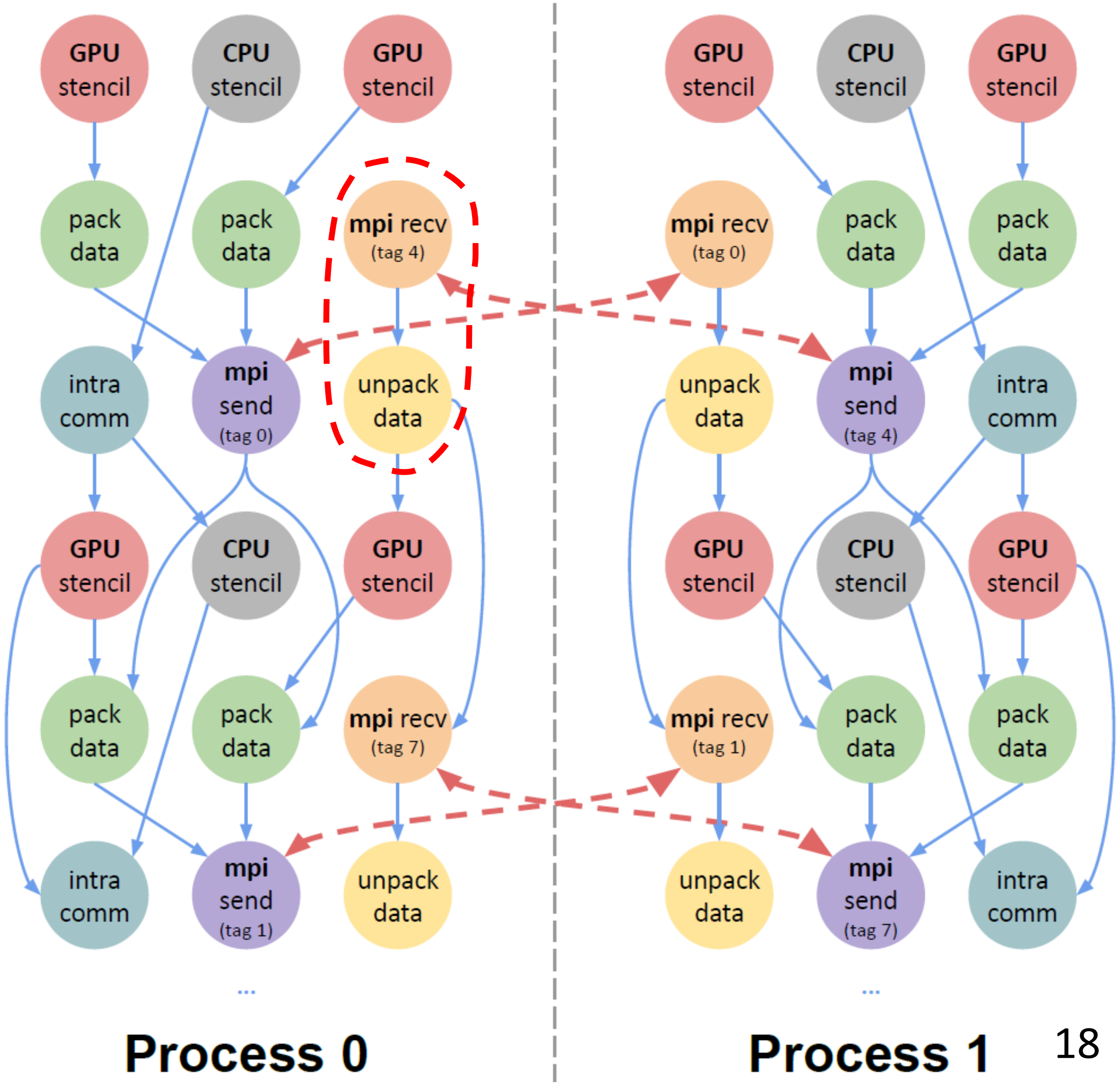


Challenges with Tasks + Operations (III)

How can we support **non-blocking** operations?

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Request req;
    MPI_Irecv(recvbuf[msg], msg_size, MPI_BYTE, src, tag, ..., &req);
    /* ?? */
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

How a task can **synchronize** with its operations?



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

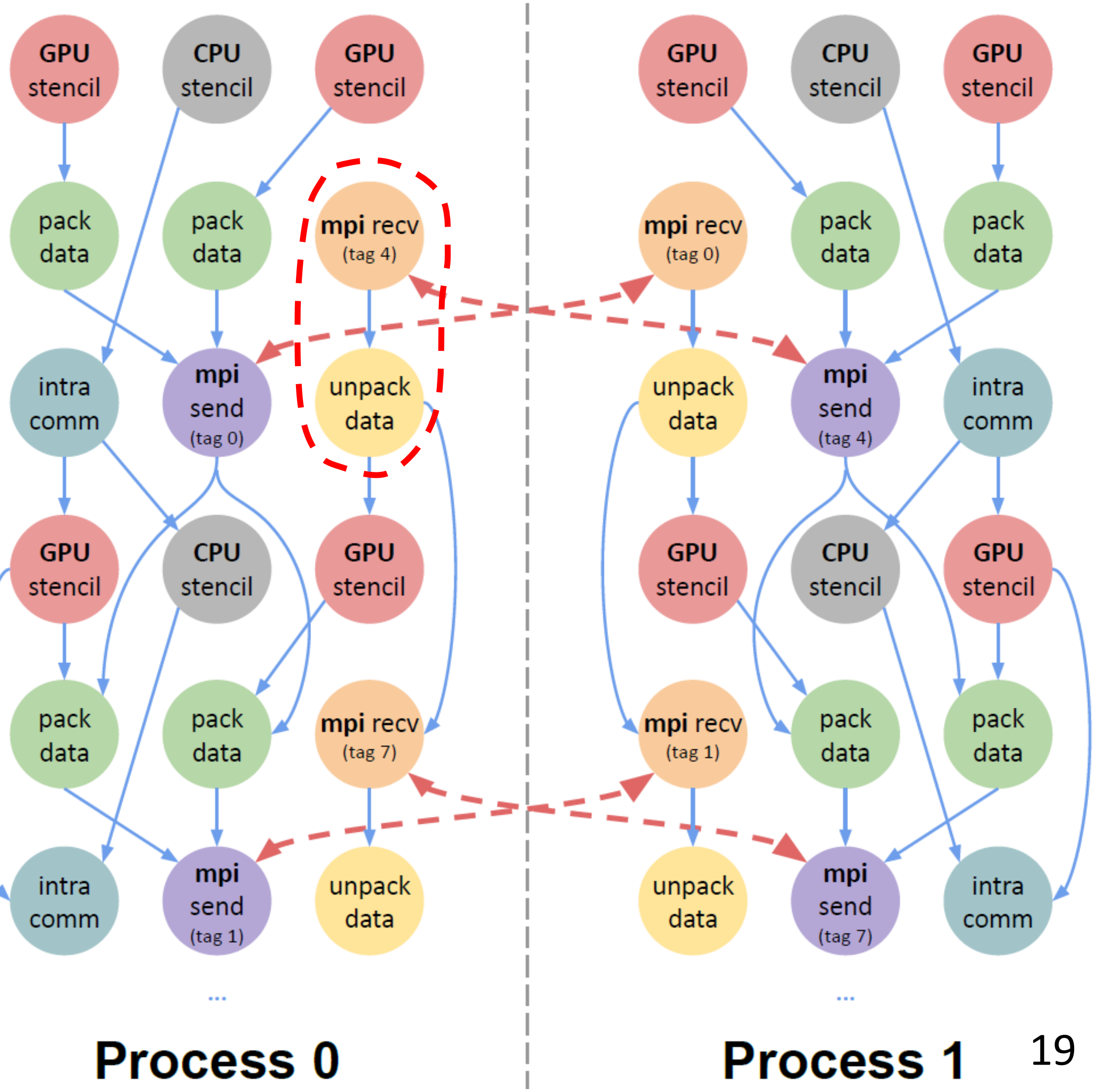
Challenges with Tasks + Operations (IV)

How can we support **non-blocking** operations?

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Request req;
    MPI_Irecv(recvbuf[msg], msg_size, MPI_BYTE, src, tag, ..., &req);
    MPI_Wait(&req, MPI_STATUS_IGNORE);
  }


  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

Blocking again?

It **detriments** the use of **non-blocking** operations…



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Principles of Task-Awareness

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outline

- Motivation
- **Principles of Task-Awareness**
  - Supporting Blocking APIs within Tasks
  - Supporting Non-Blocking APIs within Tasks
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

Principles of Task-Awareness

- **Blocking** operations
  - The function **returns after** the operation has **completed**
  - The running **thread is blocked** inside the interface (e.g., busy waiting or suspending)
  - Examples: *MPI_Recv*, *MPI_Bcast*, *cudaMemcpy*

- **Non-blocking** operations
  - The function just **issues** the operation and **returns immediately**
  - Another function is used to **check completion** later
  - Examples: *MPI_Irecv*, *MPI_Ibcast*, *cudaMemcpyAsync*

- Task-Awareness: **efficiently** call them **inside tasks**!

Outline

- Motivation
- Principles of Task-Awareness
  - **Supporting Blocking APIs within Tasks**
  - Supporting Non-Blocking APIs within Tasks
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

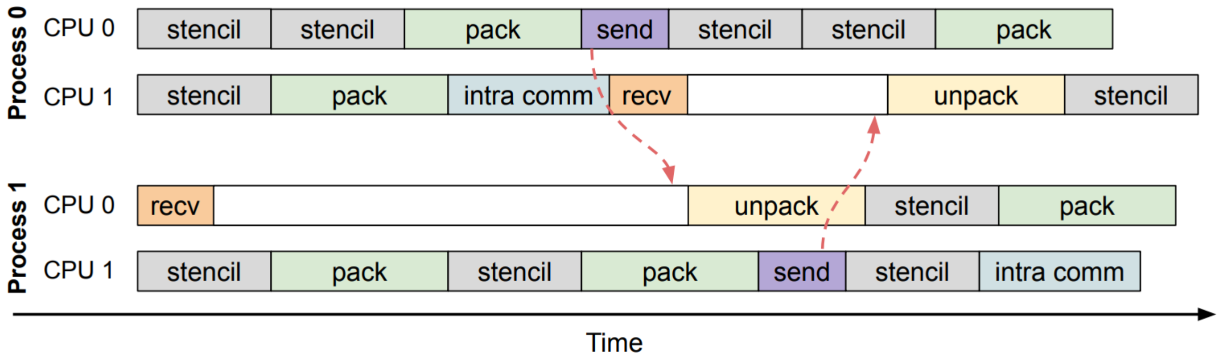
Task-Aware Blocking Operations

The task-based **runtime** is **unaware** of the idle time inside the API

- e.g., MPI or CUDA

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Recv(sendbuf[msg], msg_size, MPI_BYTE, src, tag, ...);
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

Standard Blocking Operations



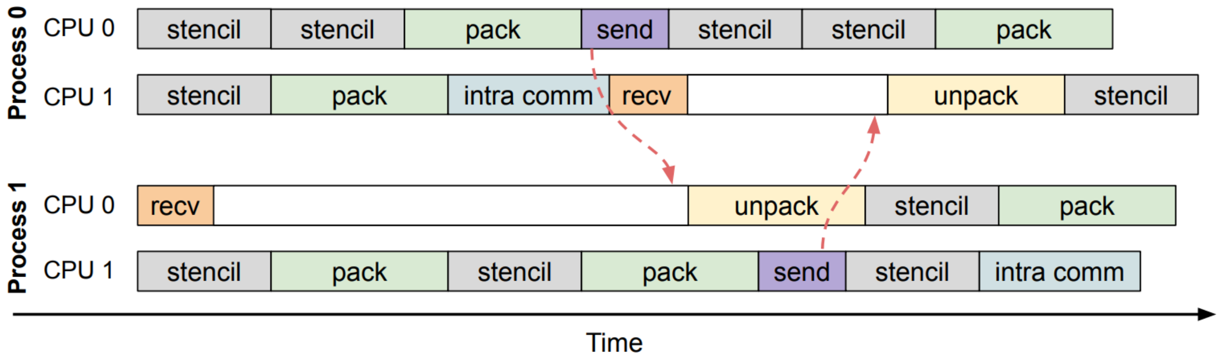
Task-Aware Blocking Operations (II)

The task-based **runtime** is **unaware** of the idle time inside the API

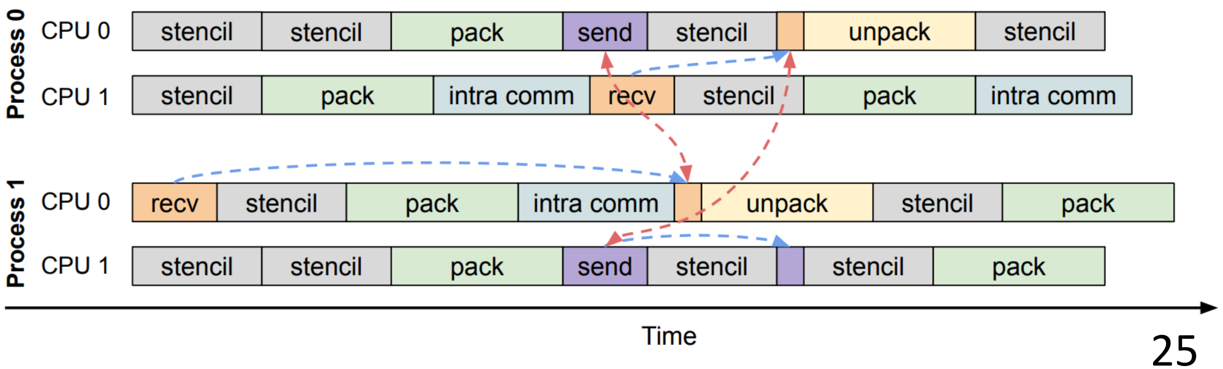
- e.g., MPI or CUDA

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Recv(sendbuf[msg], msg_size, MPI_BYTE, src, tag, ...);
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

Standard Blocking Operations



Task-Aware Blocking Operations



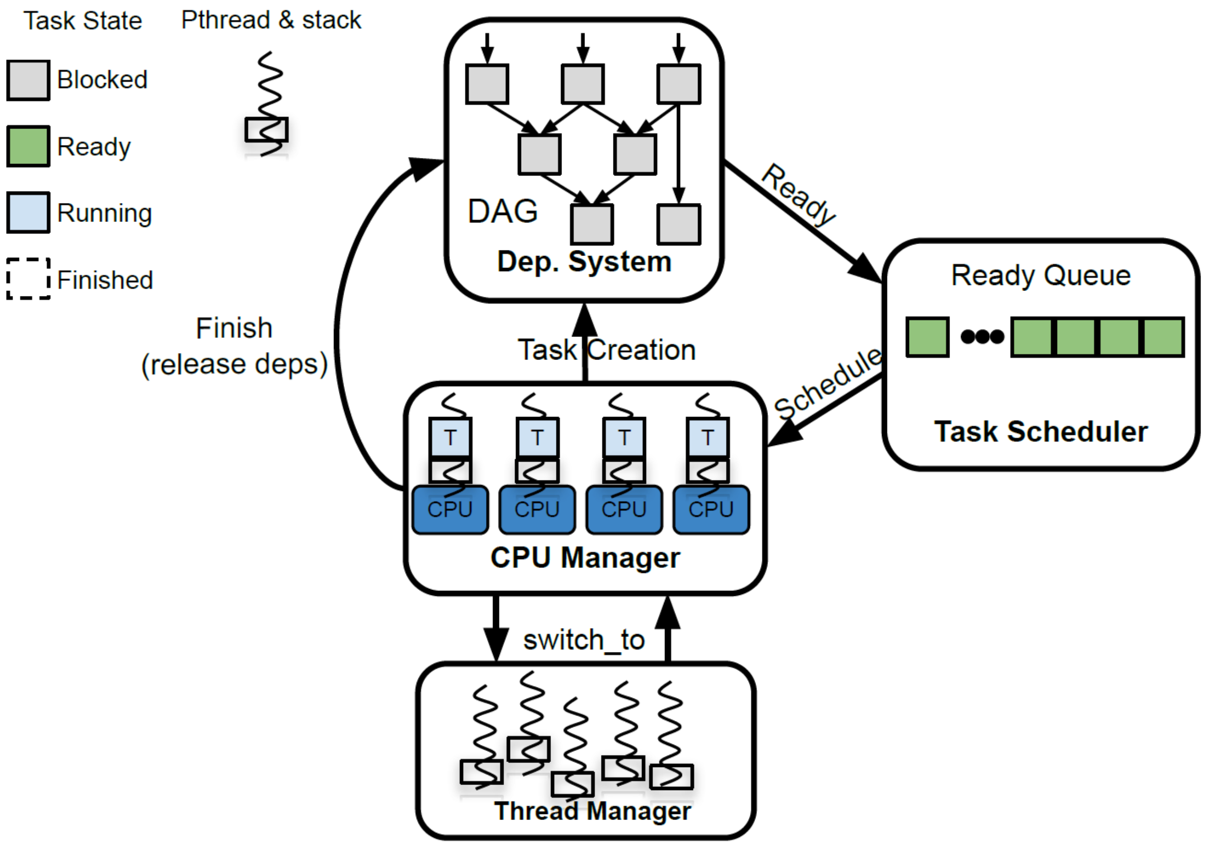
Can we **pause the task** and **yield** the **CPU**?

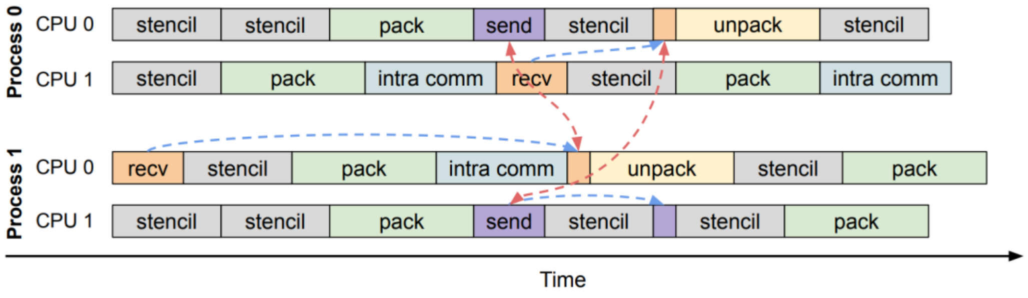
- So the runtime reuses the CPU to execute other tasks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-Aware Blocking Operations (III)

Can we **pause** the task and **yield the CPU**?

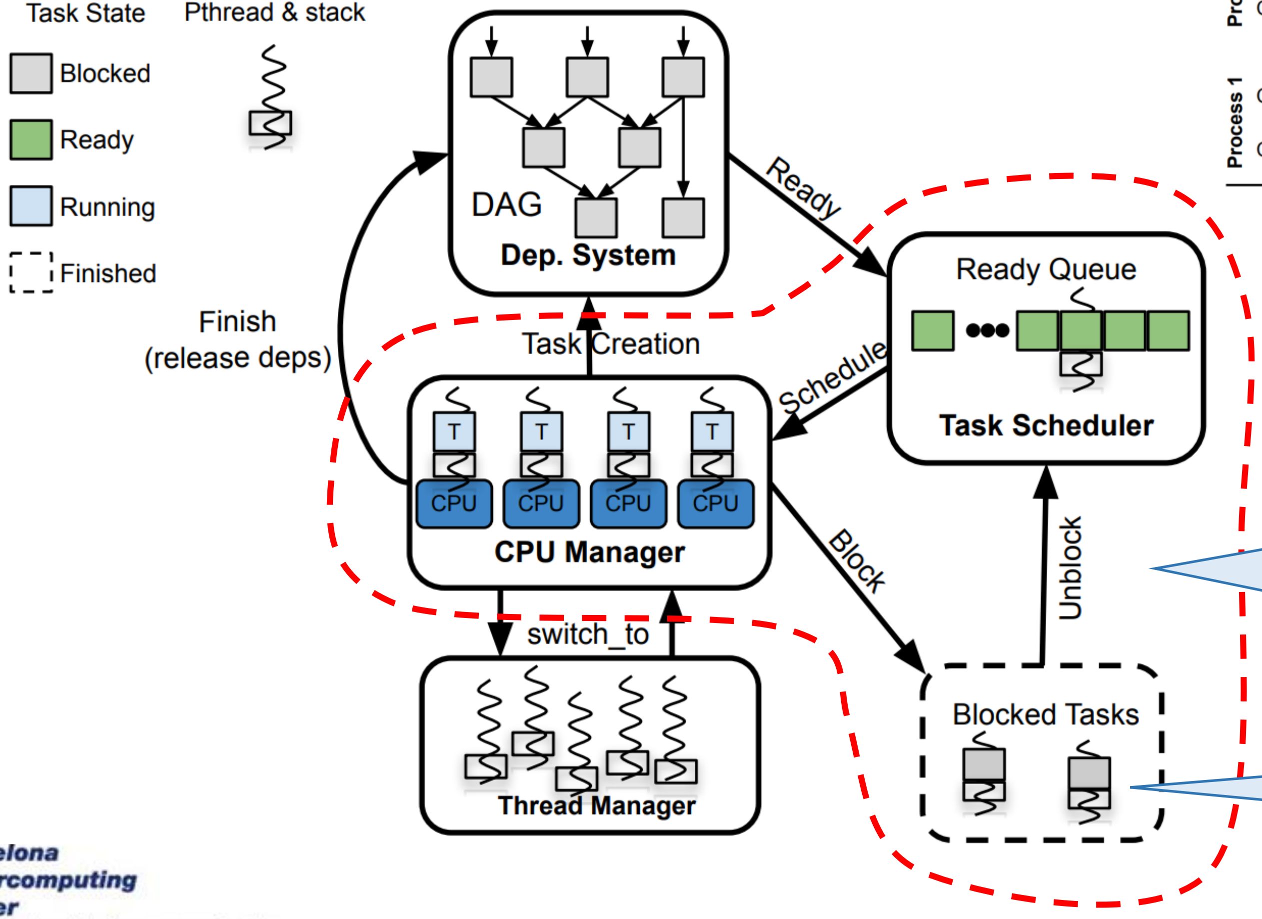


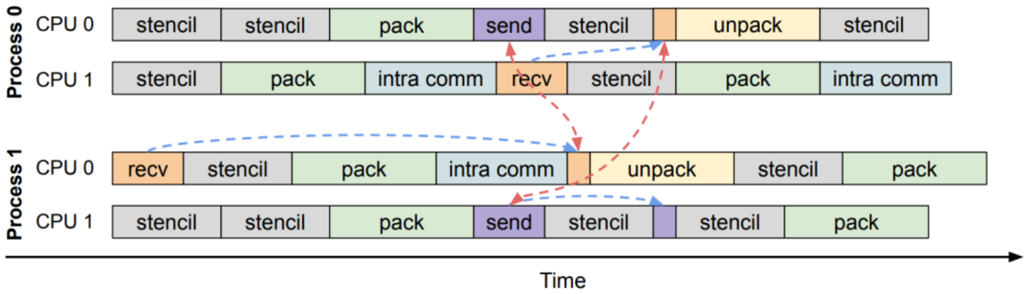


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-Aware Blocking Operations (IV)

Can we **pause** the task and **yield the CPU**?





A **polling entity** will unblock it once the operation completes

The **thread is suspended** too!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outline

- Motivation
- Principles of Task-Awareness
  - Supporting Blocking APIs within Tasks
  - **Supporting Non-Blocking APIs within Tasks**
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

Task-Aware Non-Blocking Operations

How can we **synchronize** tasks and their non-blocking operations?

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Request req;
    MPI_Irecv(recvbuf[msg], msg_size, MPI_BYTE, src, tag, ..., &req);
    /* ?? */
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

We need to **synchronize** somehow!

Task-Aware Non-Blocking Operations

How can we **synchronize** tasks and their non-blocking operations?

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Request req;
    MPI_Irecv(recvbuf[msg], msg_size, MPI_BYTE, src, tag, ..., &req);
    /* ?? */
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

But we do **not** want **blocking behavior** here!

The task does not consume the received data…

Task-Aware Non-Blocking Operations (II)

How can we **synchronize** tasks and their non-blocking operations?

```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Request req;
    MPI_Irecv(recvbuf[msg], msg_size, MPI_BYTE, src, tag, ..., &req);
    /* ?? */
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

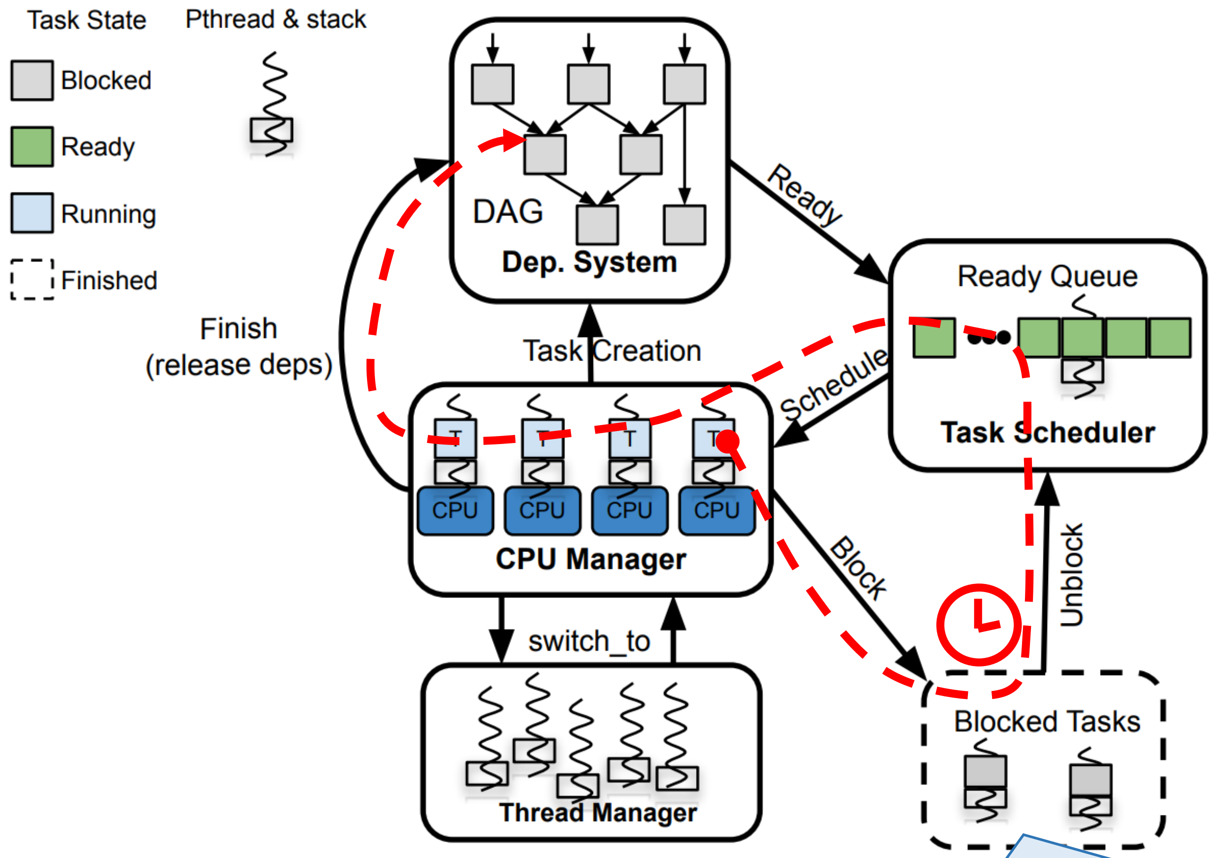
We do **not** want **blocking behavior** here!

The task does not consume the received data…

Can we just **delay** the **completion** of the **task without pausing** it?

Task-Aware Non-Blocking Operations (III)

Can we **delay** the **completion** of the **task without pausing** it?



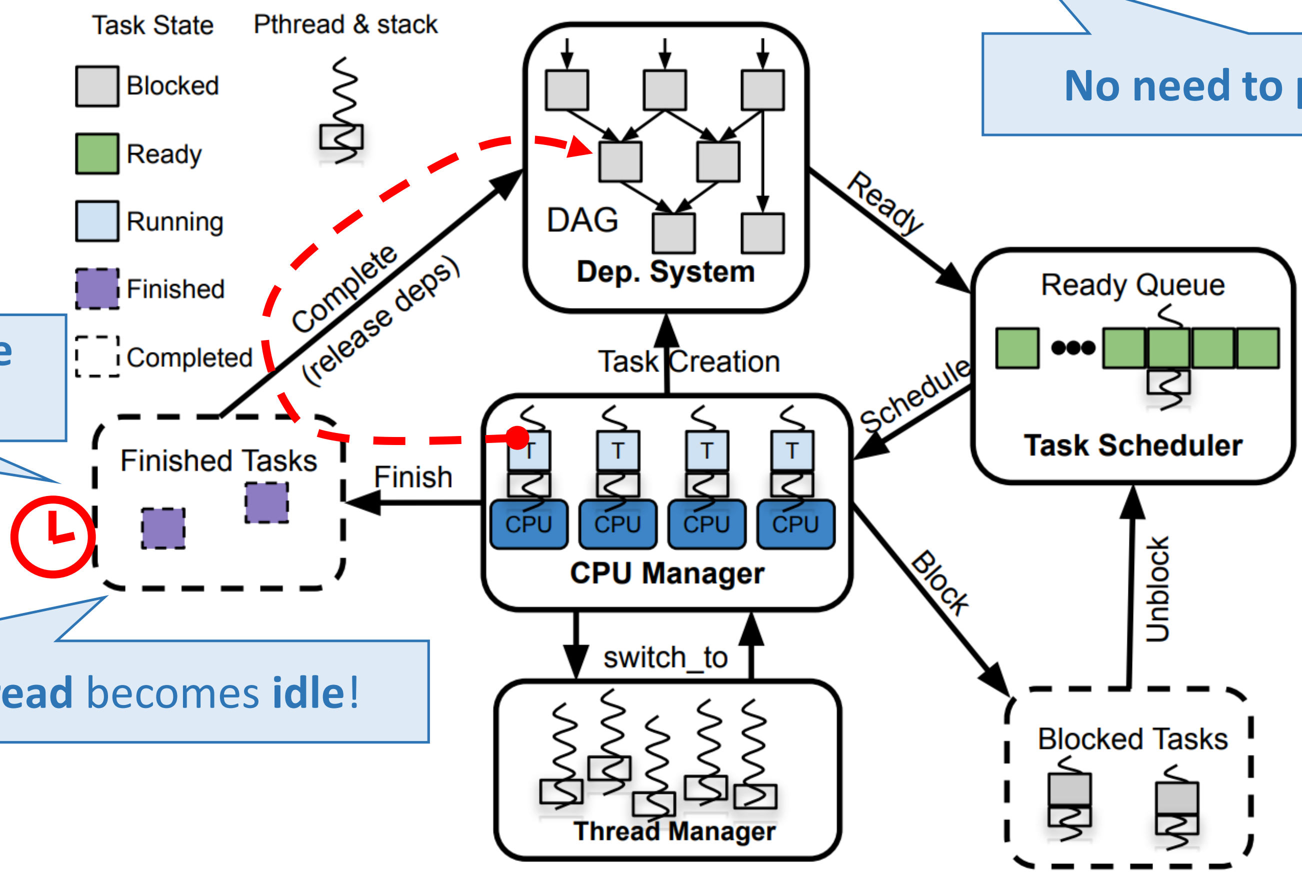
The **thread is suspended** too!

Task-Aware Non-Blocking Operations (IV)

Just **delay** the **completion** of the **task** registering **pending events**

No need to pause tasks!

Finished until its **events are fulfilled**



The **thread** becomes **idle**!

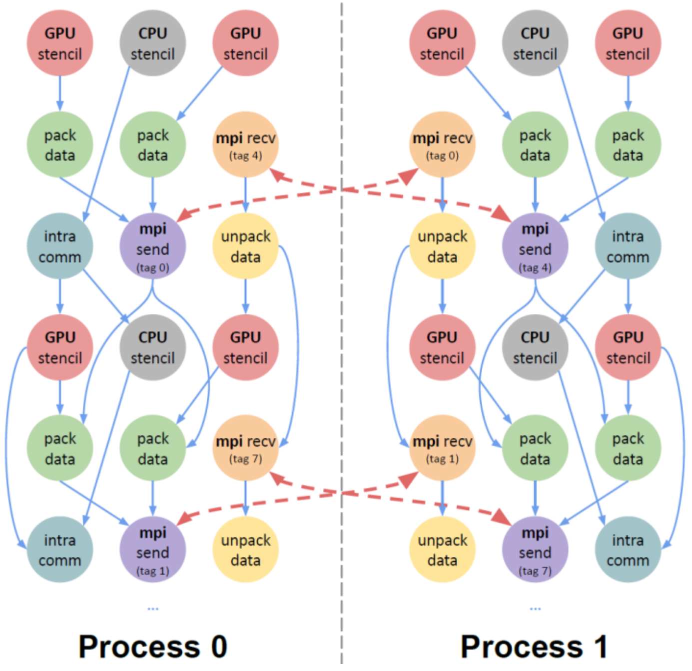
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Recap

- **Principles** for efficiently incorporating operations inside tasks
  - **Pause/resume** tasks on blocking operations
  - **Delay task completion** on non-blocking operations

- Applications will not directly use those mechanisms!

- They are the **building blocks** for our Task-Aware Libraries!
  - High-level SW solutions applying these principles transparently

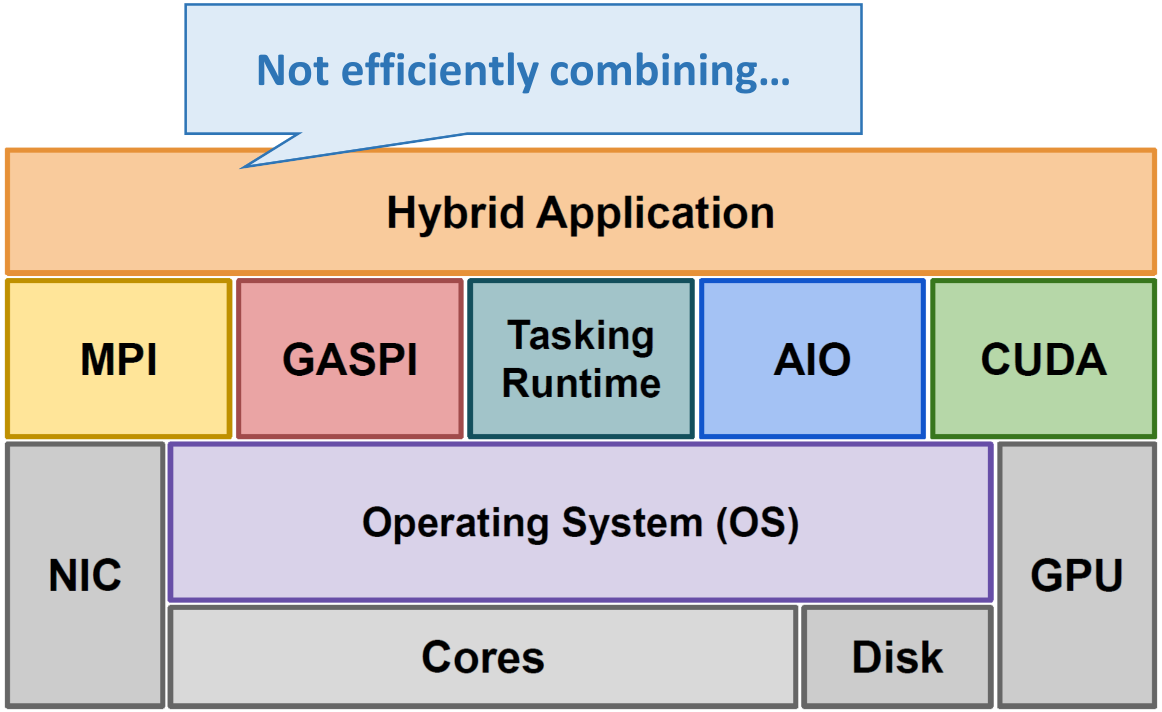


Outline

- Motivation
- Principles of Task-Awareness
- **Task-Aware Libraries (TA-X)**
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

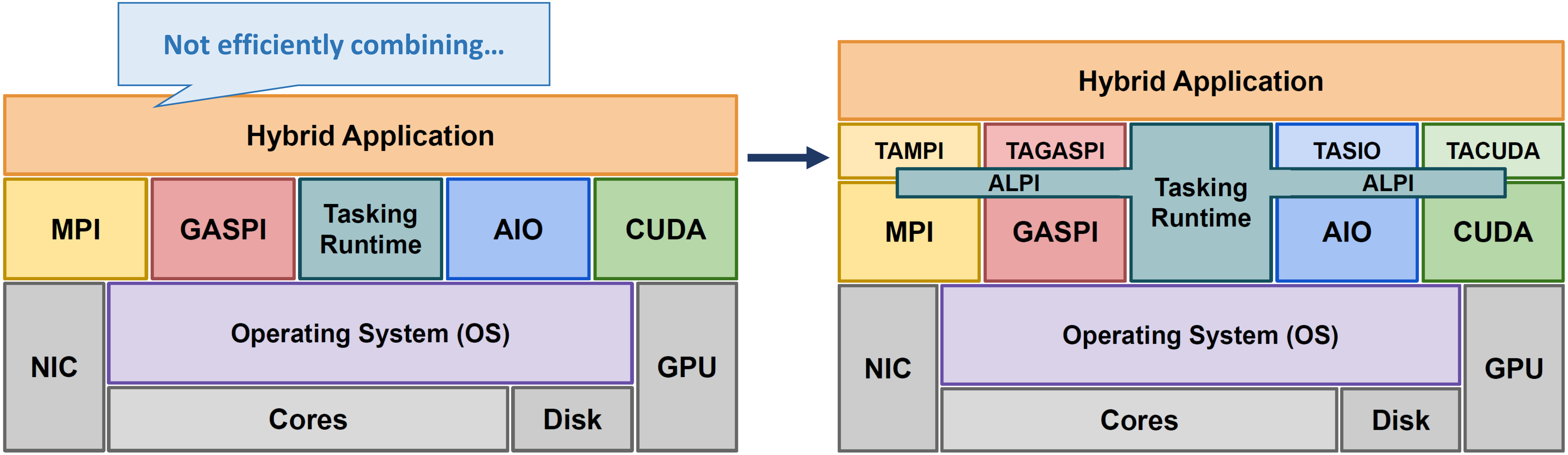
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-Aware Libraries (TA-X)



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-Aware Libraries (TA-X)



Task-Aware Libraries

- Independent libraries that allow **issuing API operations** within **tasks**
- **Implement cooperation** mechanisms between the tasking runtime and the native API

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

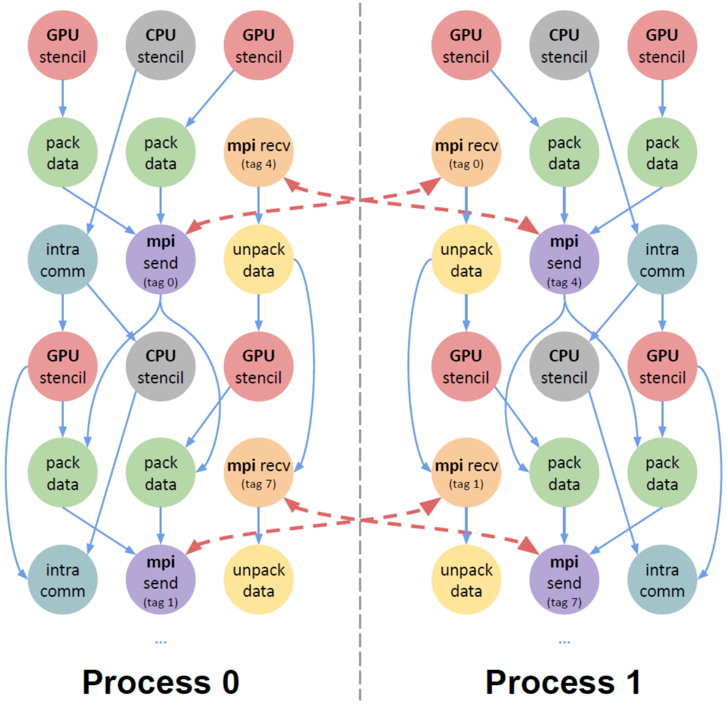
Task-Aware Libraries (TA-X)

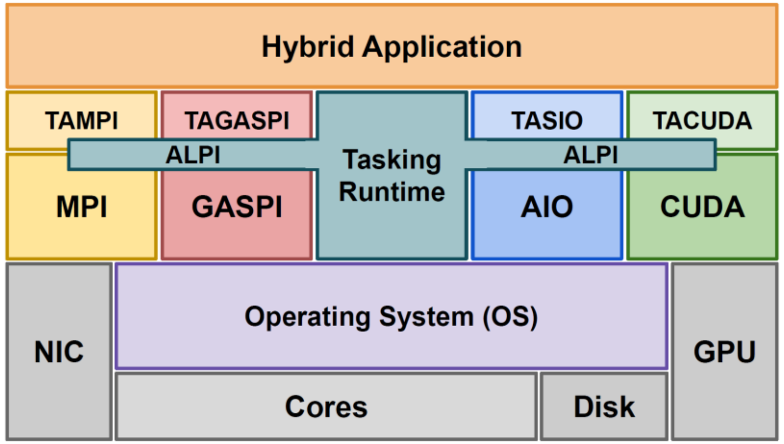
Extend the native API with **task-aware operations**

- **Blocking** operations: **pause/resume** tasks
- **Non-blocking** operations: **delay** task **completion** through events
- Guarantee **asynchronous progress**

Benefits

- **Reuse CPU resources** while operations are in-flight
- **Natural overlap** of application phases
  - Computation, communication, GPU offloading, I/O, etc.
- **Fine synchronizations** through data dependencies





Barcelona Supercomputing Center
Centro Nacional de Supercomputación

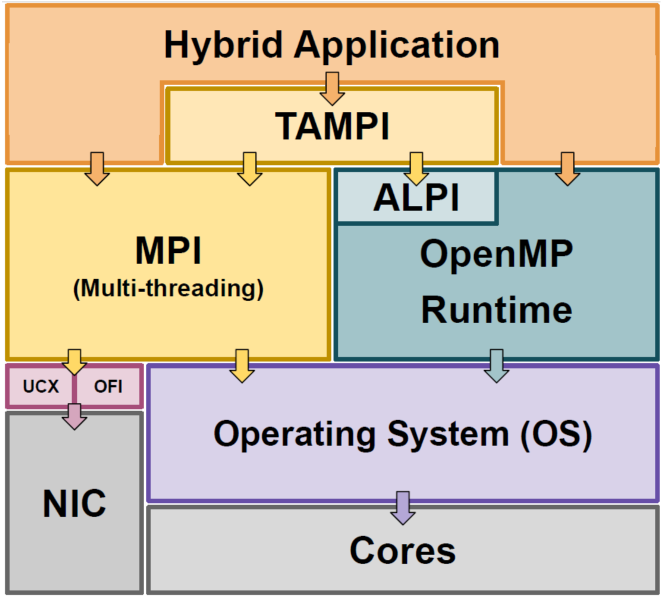
Task-Aware Libraries (TA-X)

The TA-X libraries are implemented using the **native APIs** and some **tasking services**

- **ALPI** is the low-level interface provided by the tasking runtime system

All TA-X libraries have a similar architecture

- Implementing a new TA-X library is not complex
- The interface is the same or similar to the native API



Task-Aware Libraries (TA-X)

The TA-X libraries are implemented using the **native APIs** and some **tasking services**

- **ALPI** is the low-level interface provided by the tasking runtime system

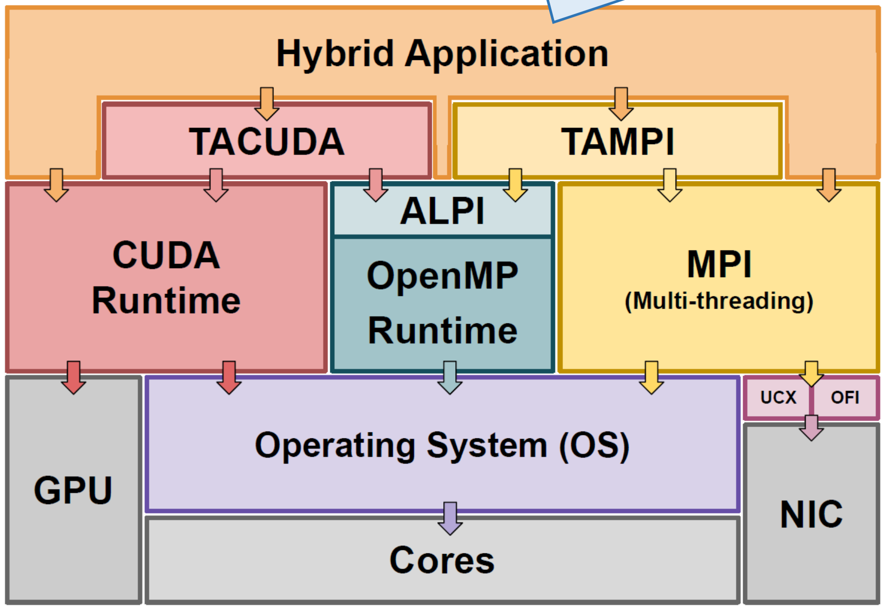
All TA-X libraries have a similar architecture

- Implementing a new TA-X library is not complex
- The interface is the same or similar to the native API

Many TA-X libraries are available

- **Task-Aware MPI (TAMPI)**
- **Task-Aware CUDA (TACUDA)**
- Task-Aware GASPI
- Task-Aware HIP
- Task-Aware SYCL
- Task-Aware AscendCL
- Task-Aware IO

We can **combine** multiple TA-X libraries!



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-Aware MPI

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Outline

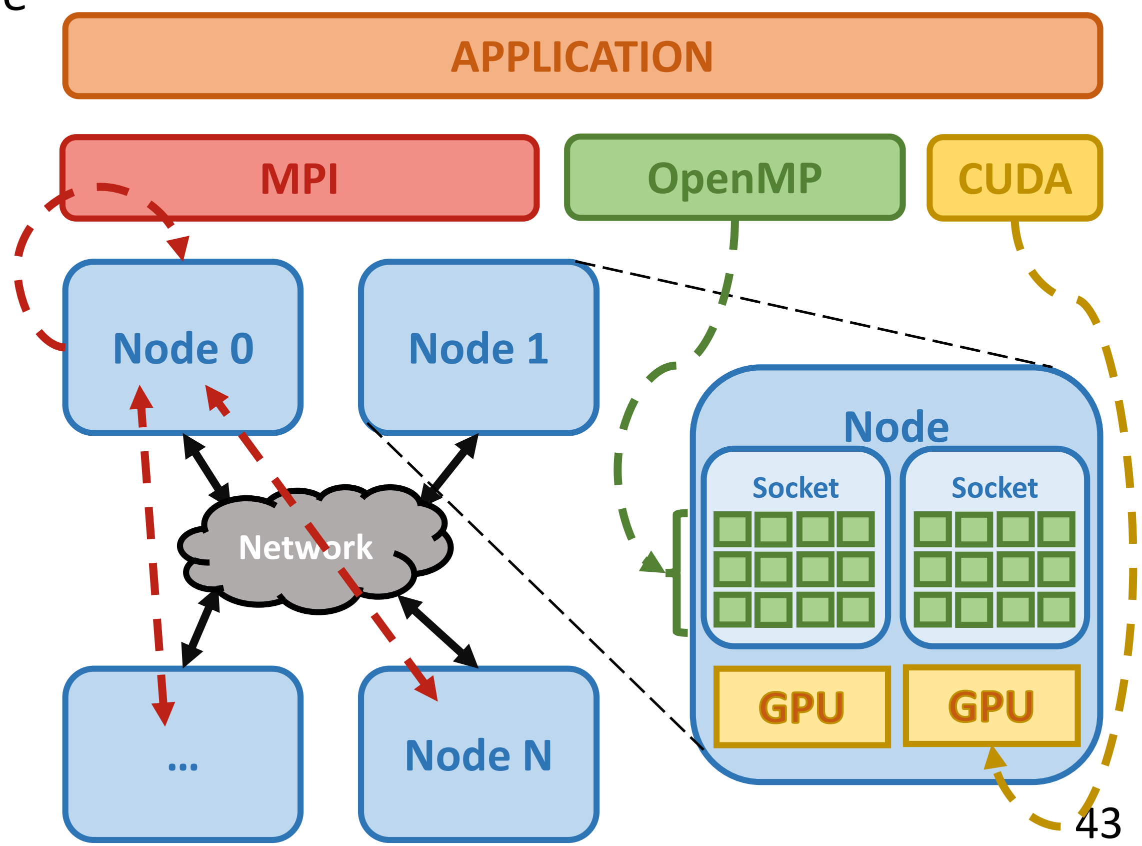
- Motivation
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- **Task-Aware MPI (TAMPI)**
  - Hybrid MPI + OpenMP Programming
  - Task-Aware MPI Library
  - Gauss-Seidel Example
  - Implementation
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

Why Hybrid Parallel Programming?

- Using MPI or OpenMP alone is not the best option

- **OpenMP-only** is not possible with more than one node
  - OpenMP is for shared-memory parallelism

- **MPI-only** can launch one MPI process per core
  - Explicit messages within the same node
  - Memory replication (per-process structures)
  - Collectives are more expensive (i.e., more processes)
  - Sensitive to system noise (e.g., preemptions)
  - Bad intra-node load balancing handling



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming distributed systems is hard

Mixing **MPI + Tasks** can bring the best of both models!

MPI is the gold standard for programming distributed HPC systems

- ✓ Good performance and scalability in distributed systems (low runtime overhead)
- ✓ Good cache and NUMA locality (SPMD model)
- ✗ Computation and communication phases have to be manually overlapped
- ✗ It is complex to mitigate load-balance issues at the node level

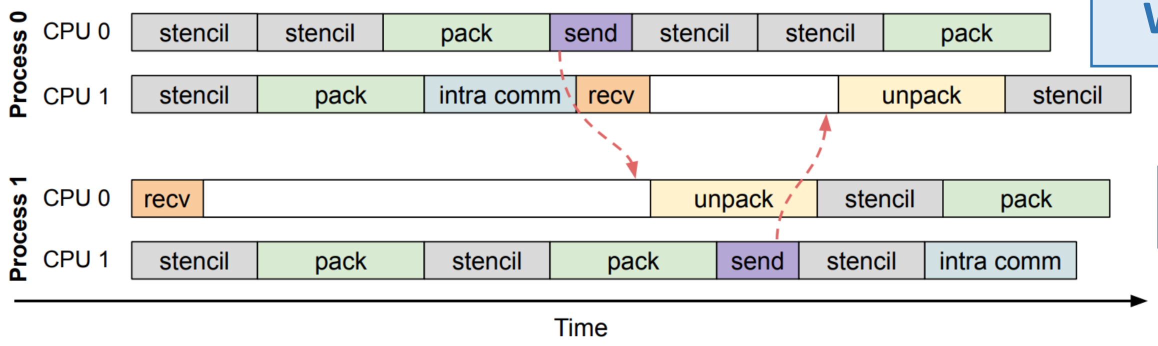
OpenMP/OmpSs-2 tasking models

- ✗ Not designed to run on distributed systems
- ✗ Bad cache and NUMA locality
- ✓ Data-flow execution model
- ✓ Natural overlap of application phases
- ✓ Good load-balancing at the node level

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hybrid Programming: MPI + OpenMP Tasks

Remember the **challenges using the standard?**



Wasting CPU resources inside MPI

Bad performance!

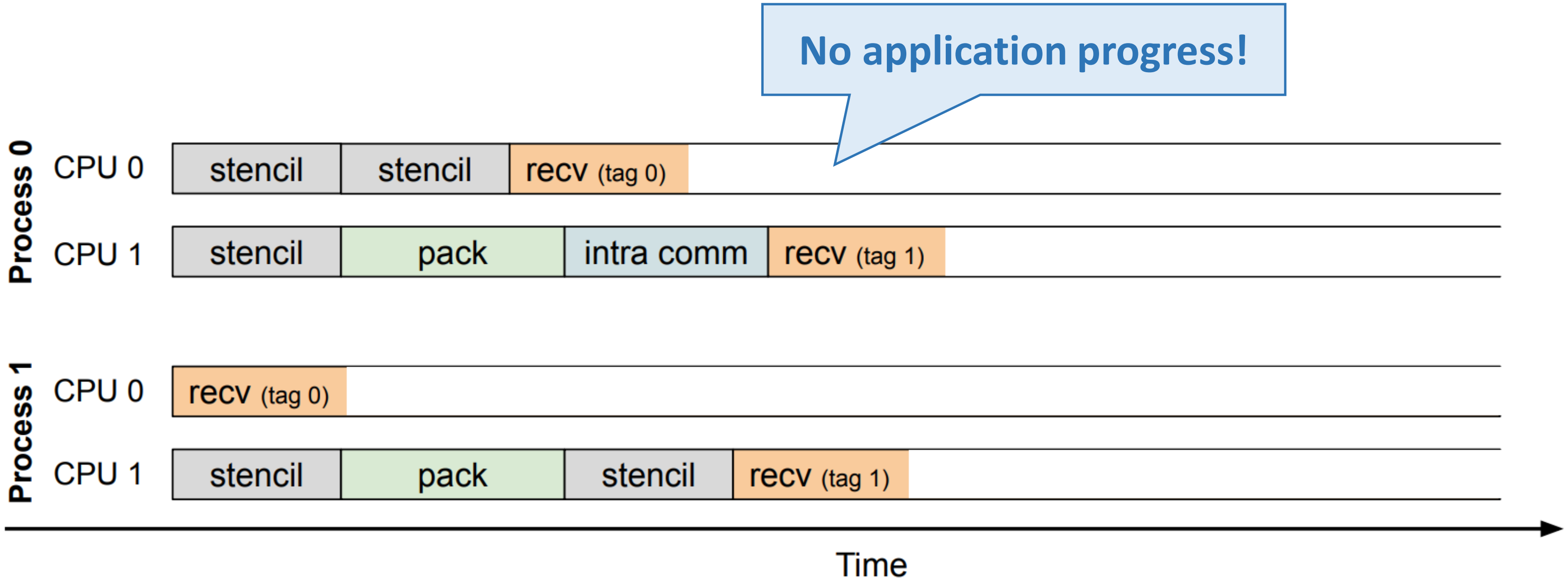
How can tasks **handle non-blocking** operations?

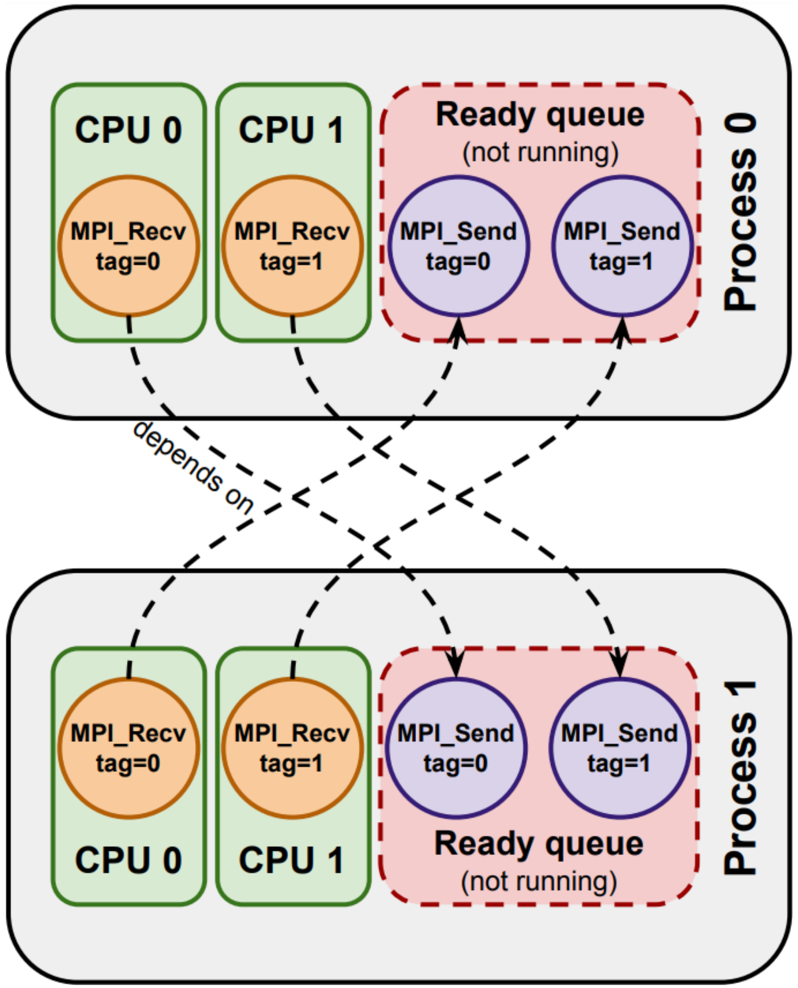
```
for (int msg = 0; msg < nmsgs; ++msg) {
  #pragma omp task depend(out: recvbuf[msg])
  {
    MPI_Request req;
    MPI_Irecv(sendbuf[msg], msg_size, MPI_BYTE, src, tag, ..., &req);
    /* ?? */
  }

  #pragma omp task depend(in: recvbuf[msg]) depend(out: data[...])
  {
    unpack_data(&data[...], recvbuf[msg], msg_size);
  }
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

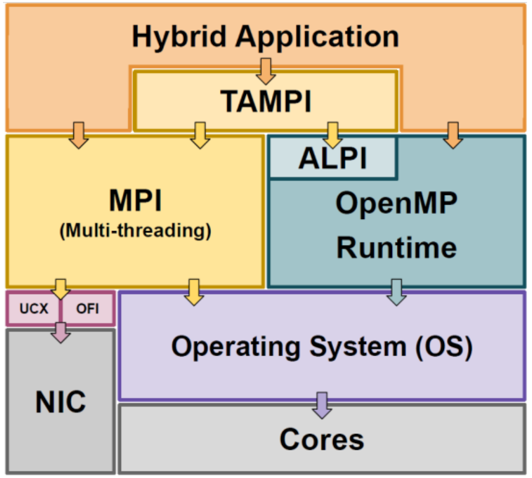
Hybrid Programming: MPI + OpenMP Tasks

Without task-awareness: **communication deadlock**!





Task-Aware MPI (TAMPI)



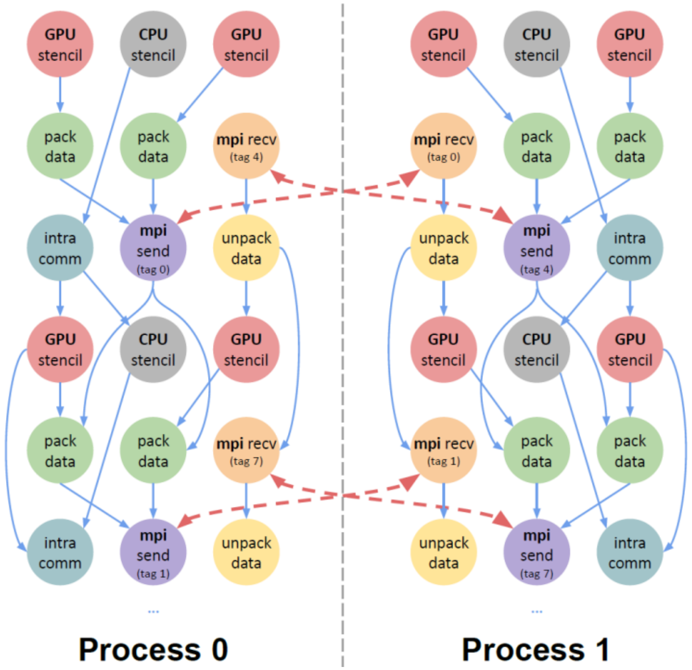
Independent library over any MPI standard library

- **Cooperation** mechanisms between the tasking runtime and MPI
- Allow incorporating **MPI** operations into the **task graphs**
- Support for **blocking** and **non-blocking** MPI operations
  - Blocking operations: **pause/resume** tasks
  - Non-blocking operations: **delay** task **completion** through events
- **Asynchronous progress**
- Support point-to-point and collectives

Built upon the task-awareness principles!

Benefits

- **Deadlock-free execution**
- CPUs are **not blocked** waiting for communication
- **Natural overlap** of computation / communication
- **Fine synchronizations** at intra- and inter-node levels



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

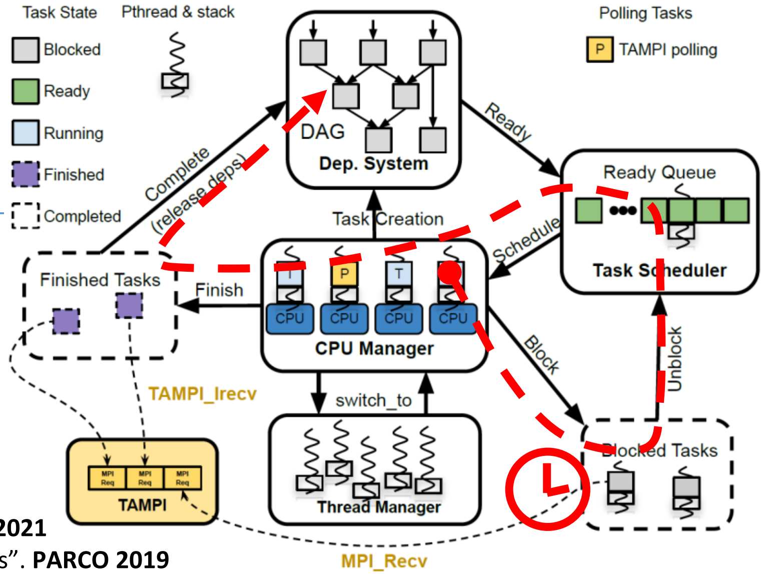
Task-Aware MPI (TAMPI)

TAMPI **Blocking** mechanism (MPI_TASK_MULTIPLE)

- **Converts standard** MPI **blocking** calls to be **task-aware**
- **Pause** the calling task while the operation is ongoing

TAMPI **intercepts** these standard functions

```
#pragma omp task depend(out: recvdata[0;nelems])
{
    MPI_Recv(recvdata, nelems, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

#pragma omp task depend(inout: bcastdata[0;nelems])
{
    MPI_Bcast(bcastdata, nelems, MPI_DOUBLE, dst, 0, MPI_COMM_WORLD);
}
```



Kevin Sala et al. "Combining One-Sided Communications with Task-Based Programming Models". **CLUSTER 2021**
Kevin Sala et al. "Integrating blocking and non-blocking MPI primitives with task-based programming models". **PARCO 2019**
Kevin Sala et al. "Towards Data-Flow Parallelization for Adaptive Mesh Refinement Applications". **CLUSTER 2020**
Kevin Sala et al. "Improving the Interoperability between MPI and Task-Based Programming Models". **EuroMPI 2018**

Task-Aware MPI (TAMPI)

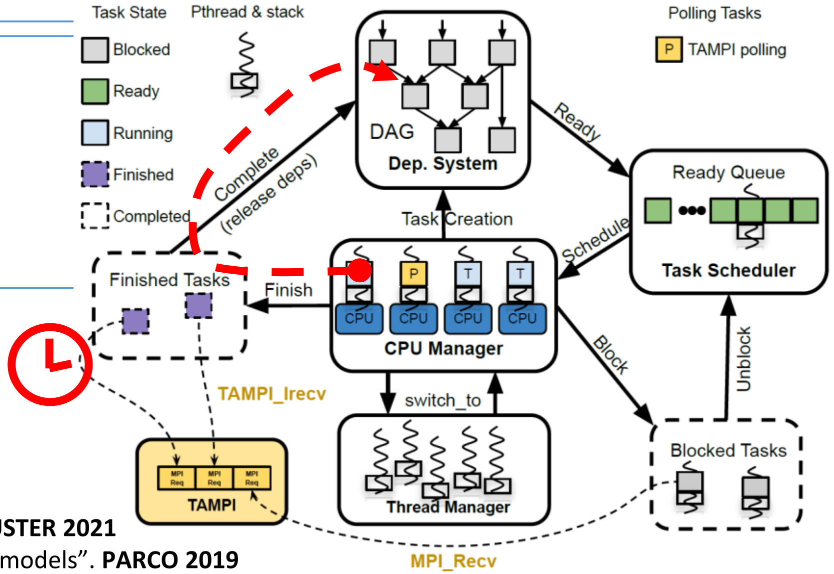
TAMPI **Non-Blocking** mechanism (MPI_THREAD_MULTIPLE)

- Provide **new** non-blocking **functions**
- **Register events** to the calling task to **delay** its **completion**

```
#pragma omp task depend(out: recvdata[0;nelems])
{
    TAMPI_Irecv(recvdata, nelems, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

OR

```
#pragma omp task depend(out: recvdata[0;nelems])
{
    MPI_Request request;
    MPI_Irecv(data, nelems, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD, &request);
    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
}
```



- Communication tasks cannot consume/reuse their buffers!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Kevin Sala et al. “Combining One-Sided Communications with Task-Based Programming Models”. **CLUSTER 2021**
Kevin Sala et al. “Integrating blocking and non-blocking MPI primitives with task-based programming models”. **PARCO 2019**
Kevin Sala et al. “Towards Data-Flow Parallelization for Adaptive Mesh Refinement Applications”. **CLUSTER 2020**
Kevin Sala et al. “Improving the Interoperability between MPI and Task-Based Programming Models”. **EuroMPI 2018**

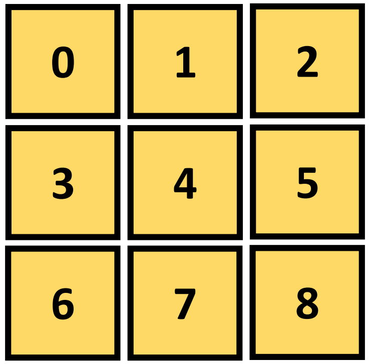
Outline

- Motivation
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
    - Hybrid MPI + OpenMP Programming
    - Task-Aware MPI Library
    - **Gauss-Seidel Example**
    - Implementation
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

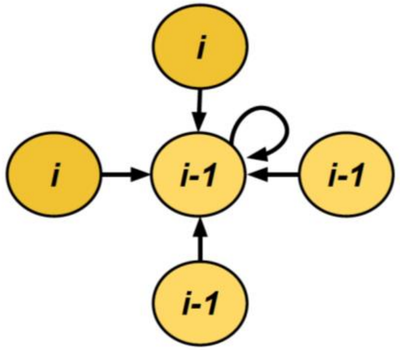
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Gauss-Seidel

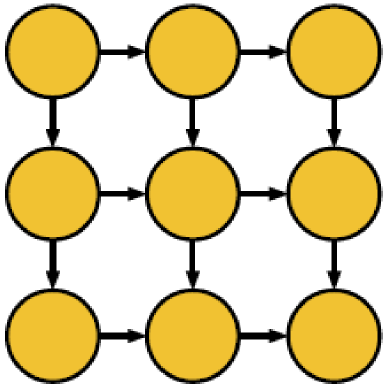
- In-place iterative algorithm over a 2D matrix and logically divided into blocks (e.g., 3 x 3 blocks domain)



- Each block depends on (1) the top and (2) the left blocks from the current iteration and (3) the right and (4) the bottom blocks from the previous iteration



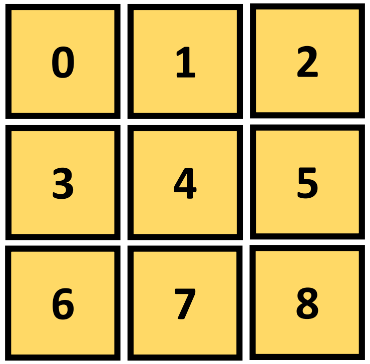
Task that computes a block on the *i-th* iteration



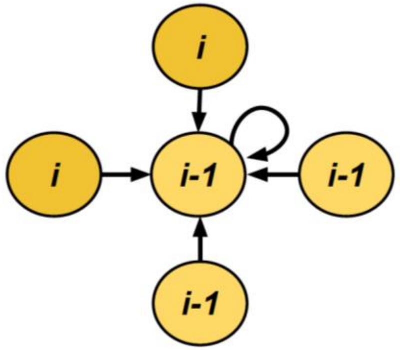
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Gauss-Seidel

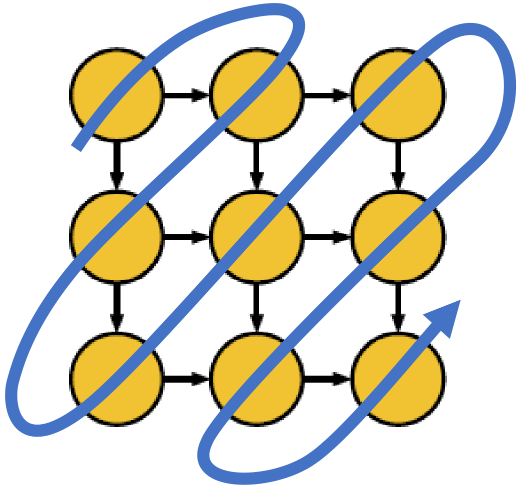
- In-place iterative algorithm over a 2D matrix and logically divided into blocks (e.g., 3 x 3 blocks domain)



- Each block depends on (1) the top and (2) the left blocks from the current iteration and (3) the right and (4) the bottom blocks from the previous iteration

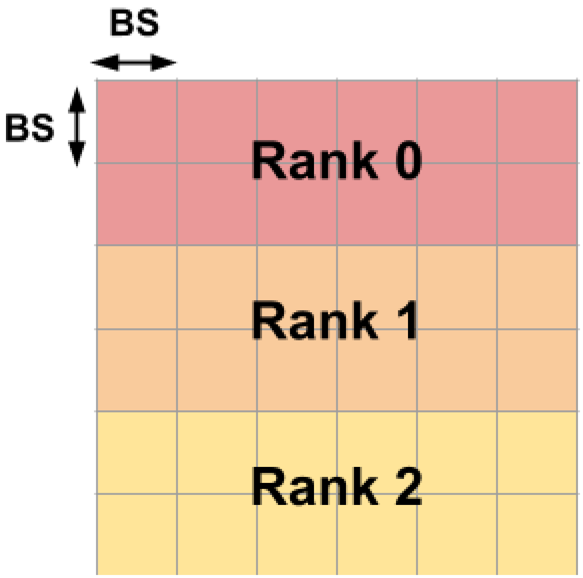


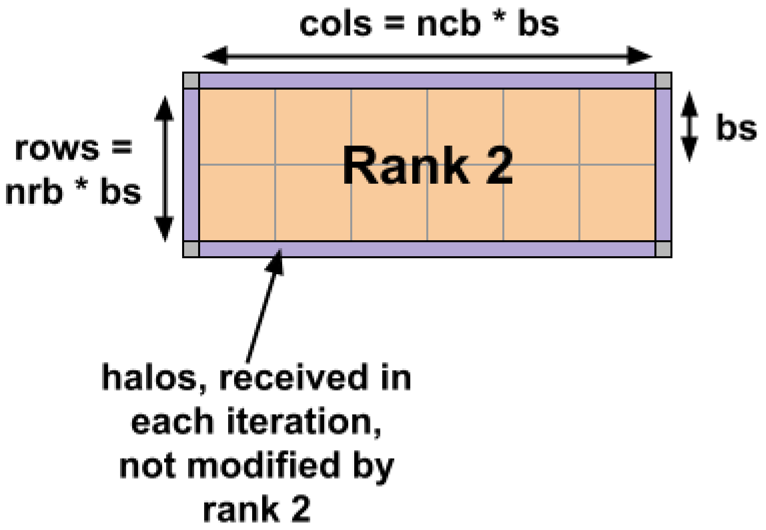
Task that computes a block on the *i-th* iteration



Gauss-Seidel: Pure MPI (II)

- Ex: 6 x 6 block domain, decomposition across 3 MPI ranks





- After each iteration, neighboring MPI ranks have to exchange upper/lower halos.

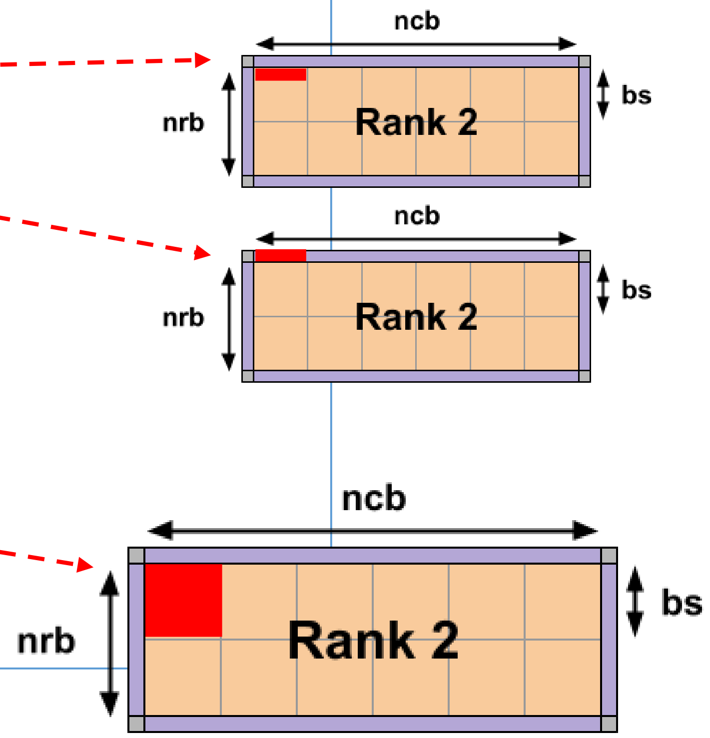
```
int nrb = # of row blocks per process
int ncb = # of column blocks
int bs =  # number of elements in each block dimension
int rows = nrb * bs + 2;
int cols = ncb * bs + 2;
double *matrix = malloc(rows * cols * sizeof(double));
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Gauss-Seidel: Pure MPI (III)

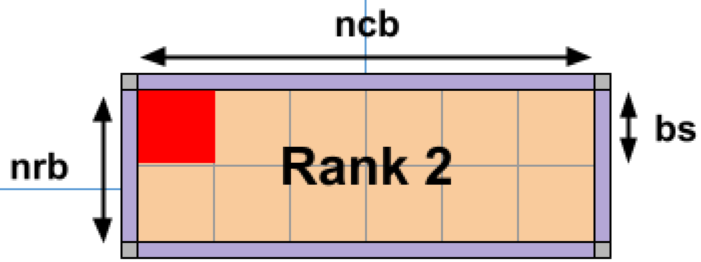
```
void solve(double matrix[rows][cols], int rows, int cols, int bs, int timesteps) {
    const int nrb = (rows-2)/bs+2;
    const int ncb = (cols-2)/bs+2;
    for (int t = 0; t < timesteps; ++t) {
        solveGaussSeidel(matrix, rows, cols, bs, nrb, ncb);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

```
void solveGaussSeidel(double matrix[rows][cols], int rows, int cols, int bs, int nrb, int ncb) {
    if (rank != 0) {
        for (int C = 1; C < ncb-1; ++C)
            send(&matrix[1][(C-1)*bs+1], bs, rank-1, 1);
        for (int C = 1; C < ncb-1; ++C)
            recv(&matrix[0][(C-1)*bs+1], bs, rank-1, 1);
    }

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            recv(&matrix[rows-1][(C-1)*bs+1], bs, rank+1, 1);

    for (int R = 1; R < nrb-1; ++R) {
        for (int C = 1; C < ncb-1; ++C) {
            computeBlock(matrix, rows, cols, (R-1)*bs+1, R*bs, (C-1)*bs+1, C*bs);

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            send(&matrix[rows-2][(C-1)*bs+1], bs, rank+1, 1);
}
```

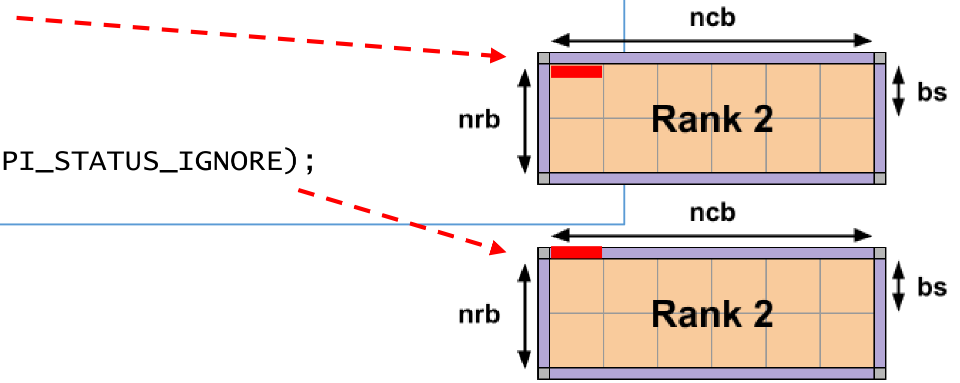


Gauss-Seidel: Pure MPI (IV)

```
void computeBlock(double M[rows][cols], int rows, int cols,
                  int rstart, int rend, int cstart, int cend) {
    for (int r = rstart; r <= rend; ++r)
        for (int c = cstart; c <= cend; ++c)
            M[r][c] = 0.25*(M[r-1][c] + M[r+1][c] + M[r][c-1] + M[r][c+1]);
}
```

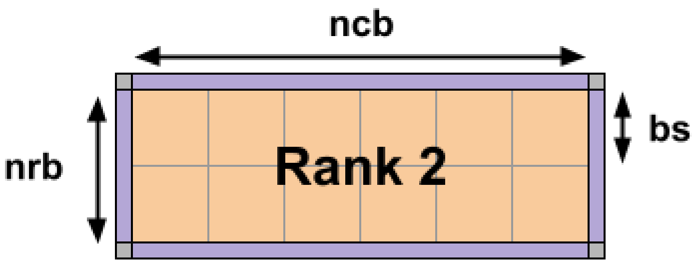



```
void send(const double *data, int nelems, int dst, int tag) {
    MPI_Send(data, nelems, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD);
}

void recv(double *data, int nelems, int src, int tag) {
    MPI_Recv(data, nelems, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

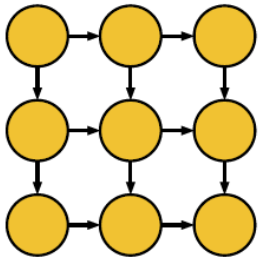


Gauss-Seidel: Pure MPI (V)

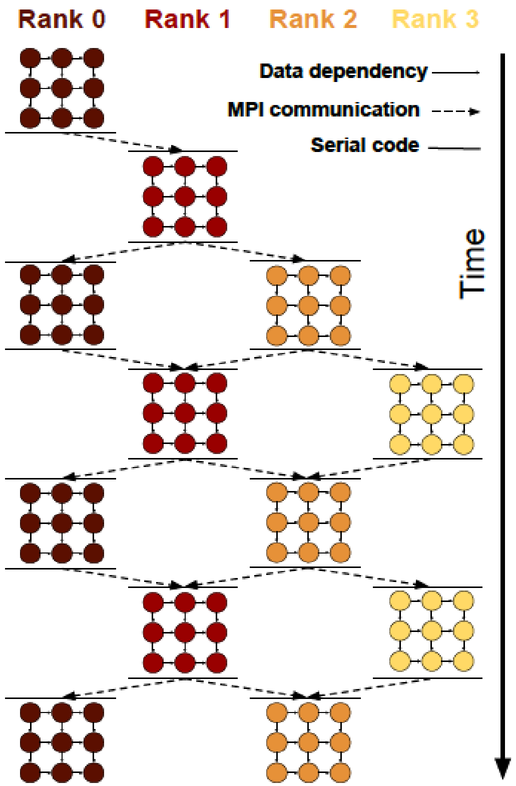
- Fully sequential execution
- **No overlapping** of communication and computation phases
- **One rank per core**




```
void solveGaussSeidel(double matrix[rows][cols], int rows, int cols, int bs, int nrb, int ncb) {
    if (rank != 0) {
        for (int C = 1; C < ncb-1; ++C)
            send(&matrix[1][(C-1)*bs+1], bs, rank-1, 1);
        for (int C = 1; C < ncb-1; ++C)
            recv(&matrix[0][(C-1)*bs+1], bs, rank-1, 1);
    }

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            recv(&matrix[rows-1][(C-1)*bs+1], bs, rank+1, 1);

    for (int R = 1; R < nrb-1; ++R) {
        for (int C = 1; C < ncb-1; ++C) {
            computeBlock(matrix, rows, cols, (R-1)*bs+1, R*bs, (C-1)*bs+1, C*bs);

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            send(&matrix[rows-2][(C-1)*bs+1], bs, rank+1, 1);
}
```

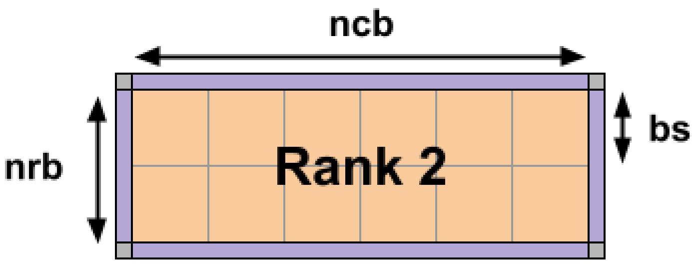


MPI_THREAD_SINGLE

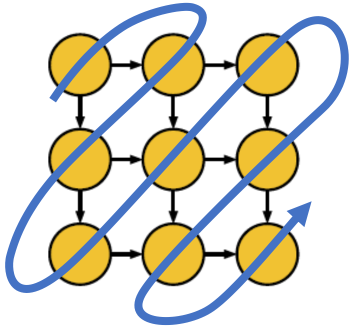


Tiled Gauss-Seidel: Fork-Join

- Parallelize **computation** phase with OmpSs-2 **tasks**
- **No overlapping** of communication and computation phases
- **One rank per socket (or node)**

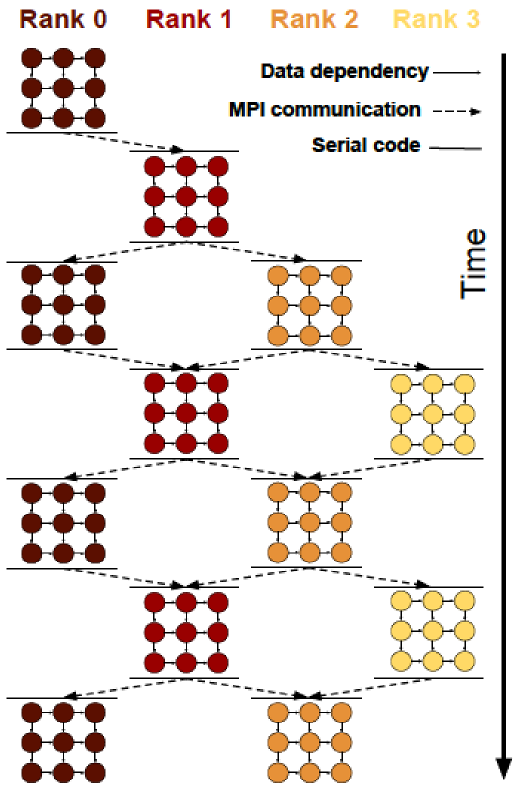



```
void solveGaussSeidel(double matrix[rows][cols], int rows, int cols, int bs, int nrb, int ncb) {
    if (rank != 0) {
        for (int C = 1; C < ncb-1; ++C)
            send(&matrix[1][(C-1)*bs+1], bs, rank-1, 1);
        for (int C = 1; C < ncb-1; ++C)
            recv(&matrix[0][(C-1)*bs+1], bs, rank-1, 1);
    }

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            recv(&matrix[rows-1][(C-1)*bs+1], bs, rank+1, 1);

    for (int R = 1; R < nrb-1; ++R) {
        for (int C = 1; C < ncb-1; ++C) {
            #pragma omp task depend(inout: matrix[bx][by]) \
                             depend(in: matrix[bx-1][by], matrix[bx][by-1]) \
                             depend(in: matrix[bx][by+1], matrix[bx+1][by])
            computeBlock(matrix, rows, cols, (R-1)*bs+1, R*bs, (C-1)*bs+1, C*bs);

    #pragma omp taskwait

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            send(&matrix[rows-2][(C-1)*bs+1], bs, rank+1, 1);
}
```



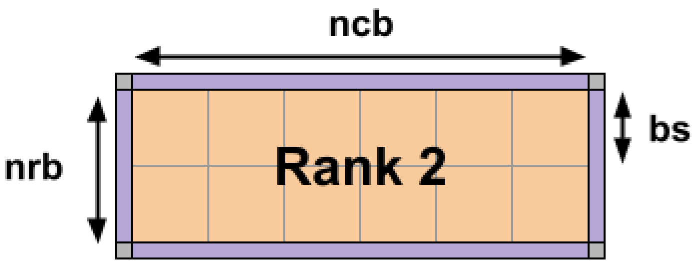
MPI_THREAD_FUNNELED

Communications are sequential



Gauss-Seidel: Tasks + Sentinel

- Parallelize **computation** and **communications** with OmpSs-2 **tasks**
- **Communication tasks** have to be <u>**serialized**</u> to avoid **deadlocks**
- **Partial overlapping** of communication and computation phases



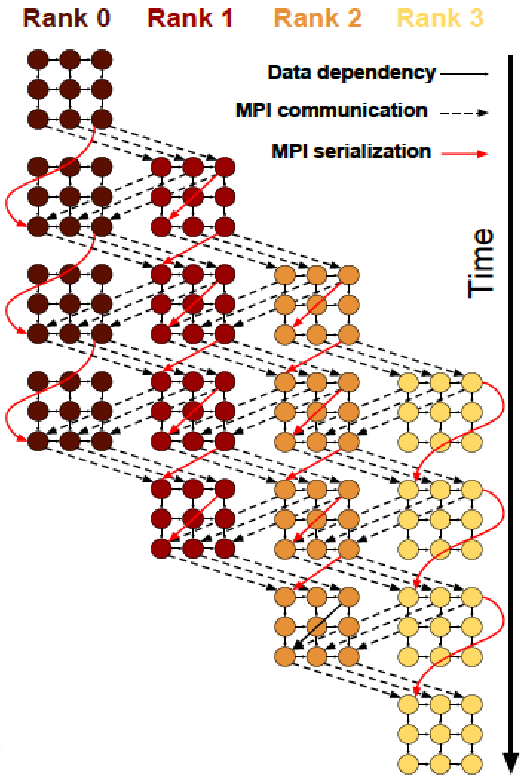

```
void solveGaussSeidel(double matrix[rows][cols], int rows, int cols, int bs, int nrb, int ncb) {
    if (rank != 0) {
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(in: reps[1][C]) depend(inout: serial)
            send(&matrix[1][(C-1)*bs+1], bs, rank-1, 1);
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(out: reps[0][C]) depend(inout: serial)
            recv(&matrix[0][(C-1)*bs+1], bs, rank-1, 1);
    }

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(out: reps[nrb-1][C]) depend(inout: serial)
            recv(&matrix[rows-1][(C-1)*bs+1], bs, rank+1, 1);

    for (int R = 1; R < nrb-1; ++R) {
        for (int C = 1; C < ncb-1; ++C) {
            #pragma omp task depend(inout: matrix[bx][by]) \
                             depend(in: matrix[bx-1][by], matrix[bx][by-1]) \
                             depend(in: matrix[bx][by+1], matrix[bx+1][by])
            computeBlock(matrix, rows, cols, (R-1)*bs+1, R*bs, (C-1)*bs+1, C*bs);

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(in: reps[nrb-2][C]) depend(inout: serial)
            send(&matrix[rows-2][(C-1)*bs+1], bs, rank+1, 1);
}
```

Sentinel to **serialize** communication tasks

Can use same message **tag**

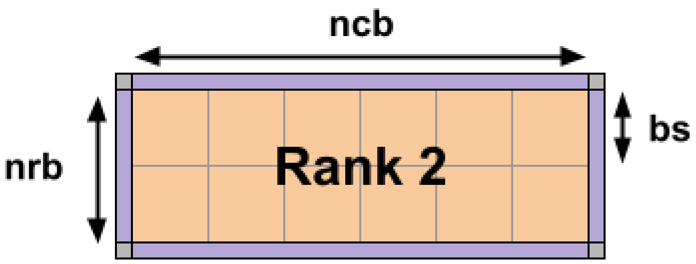
Remove taskwait!

MPI_THREAD_MULTIPLE



Gauss-Seidel: Tasks + Blocking TAMPI

- Parallelize **computation** and **communications** with OmpSs-2 **tasks**
- **Tags (block id)** used to match send and receive operations
- **Full overlapping** of communication and computation phases

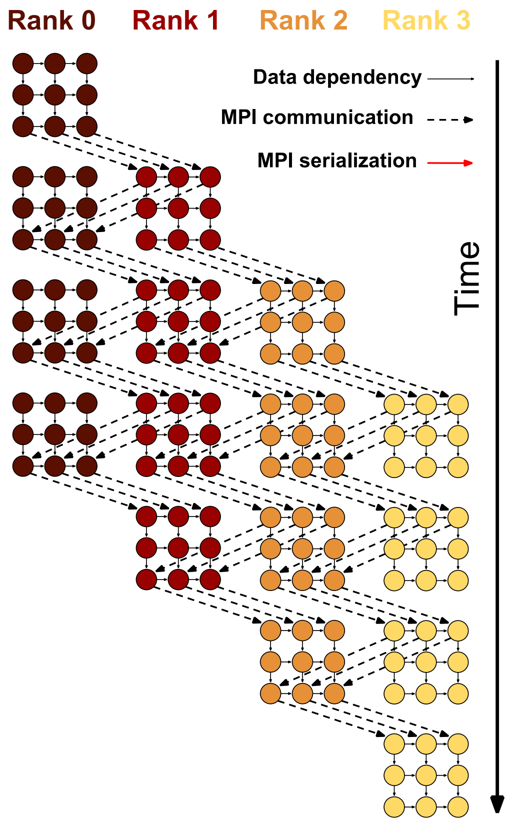



```
void solveGaussSeidel(double matrix[rows][cols], int rows, int cols, int bs, int nrb, int ncb) {
    if (rank != 0) {
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(in: reps[1][C])
            send(&matrix[1][(C-1)*bs+1], bs, rank-1, C);
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(out: reps[0][C])
            recv(&matrix[0][(C-1)*bs+1], bs, rank-1, C);
    }

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(out: reps[nrb-1][C])
            recv(&matrix[rows-1][(C-1)*bs+1], bs, rank+1, C);

    for (int R = 1; R < nrb-1; ++R) {
        for (int C = 1; C < ncb-1; ++C) {
            #pragma omp task depend(inout: matrix[bx][by]) \
                             depend(in: matrix[bx-1][by], matrix[bx][by-1]) \
                             depend(in: matrix[bx][by+1], matrix[bx+1][by])
            computeBlock(matrix, rows, cols, (R-1)*bs+1, R*bs, (C-1)*bs+1, C*bs);

    if (rank != nranks-1)
        for (int C = 1; C < ncb-1; ++C)
            #pragma omp task depend(in: reps[nrb-2][C])
            send(&matrix[rows-2][(C-1)*bs+1], bs, rank+1, C);
}
```

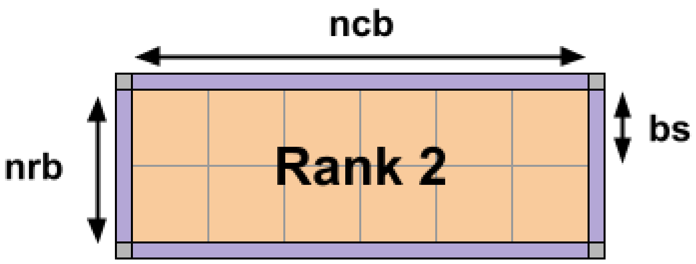
No serialization needed!

Start using block id as the message **tag**

MPI_TASK_MULTIPLE



Gauss-Seidel: Tasks + Blocking TAMPI (II)

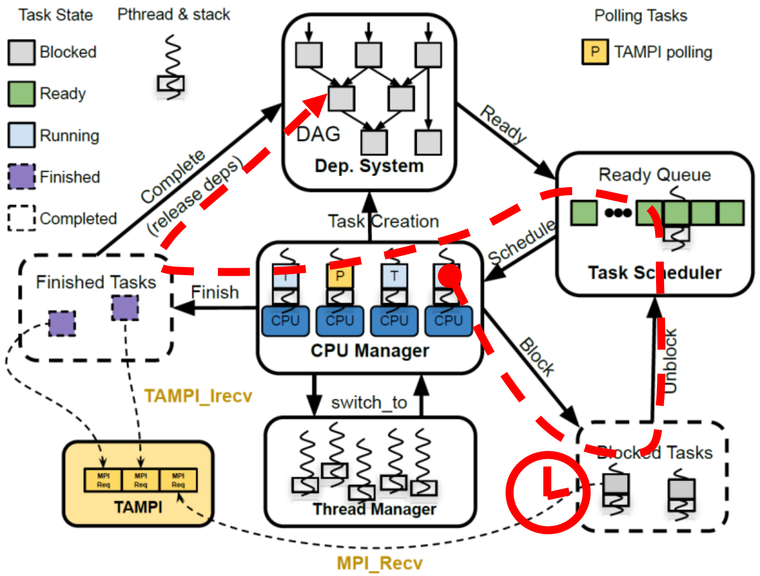


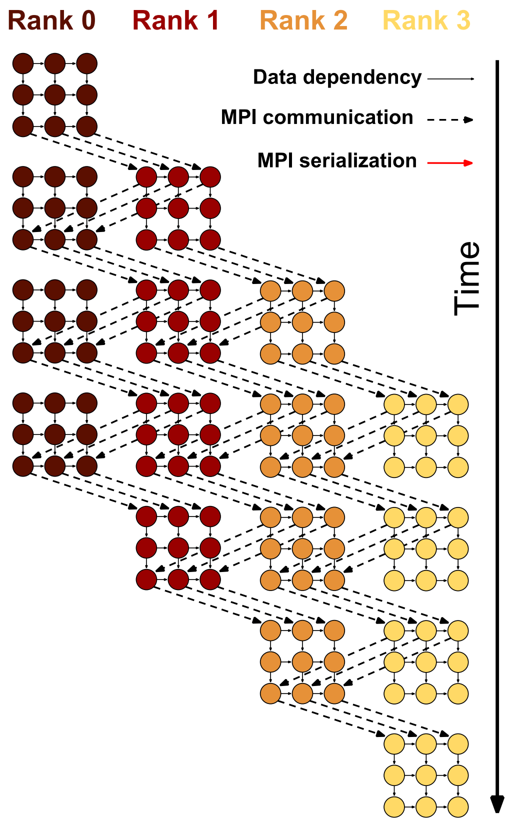
- Parallelize **computation** and **communications** with OmpSs-2 **tasks**
- **Tags (block id)** used to match send and receive operations
- **Full overlapping** of communication and computation phases
- One rank per socket (or node)

```
void send(const double *data, int nelems, int dst, int tag) {
    MPI_Send(data, nelems, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD);
}

void recv(double *data, int nelems, int src, int tag) {
    MPI_Recv(data, nelems, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

MPI_TASK_MULTIPLE

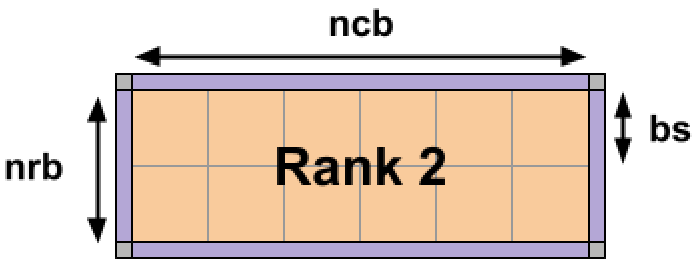
TAMPI applies **task-awareness** to all **blocking MPI calls**!





Gauss-Seidel: Tasks + Non-Blocking TAMPI

- Parallelize **computation** and **communications** with OmpSs-2 **tasks**
- **Tags (block id)** used to match send and receive operations
- **Full overlapping** of communication and computation phases
- One rank per socket (or node)



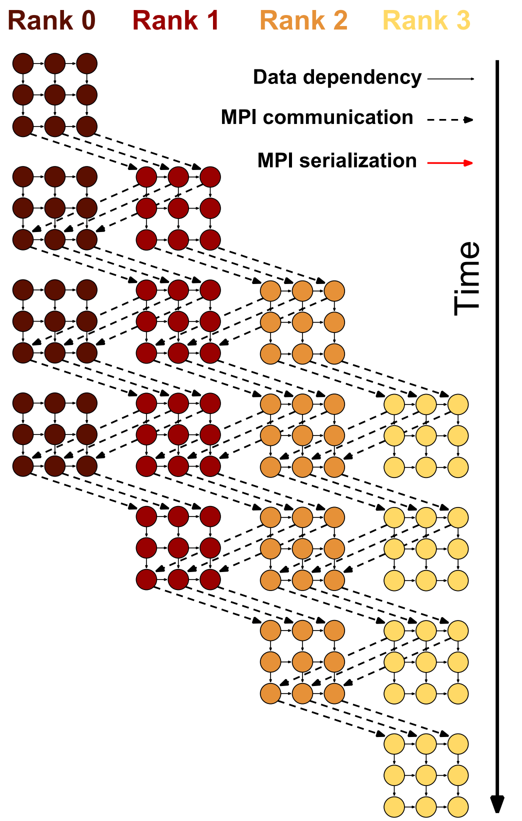

```
void send(const double *data, int nelems, int dst, int tag) {
  TAMPI_Isend(data, nelems, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

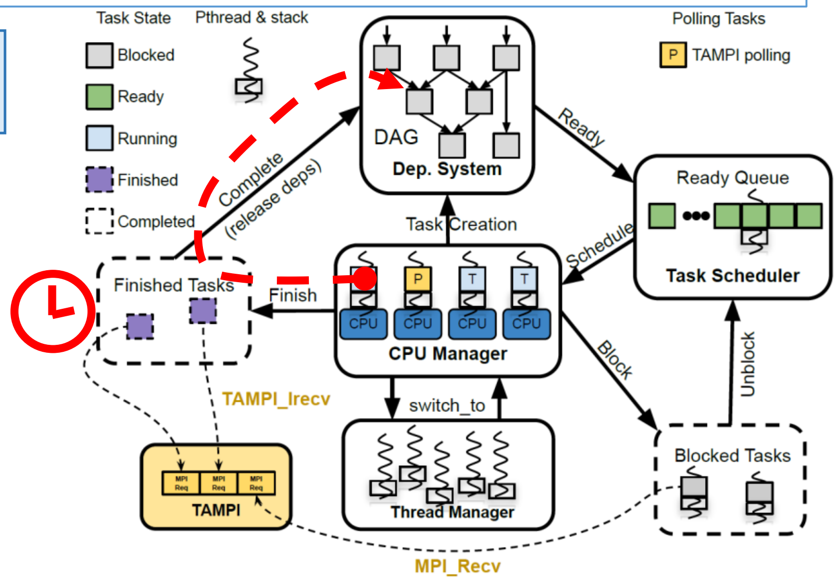
OR

```
void send(const double *data, int nelems, int dst, int tag) {
    MPI_Request request;
    MPI_Isend(data, nelems, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD, &request);
    TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
}
```

MPI_THREAD_MULTIPLE

Regular non-blocking send





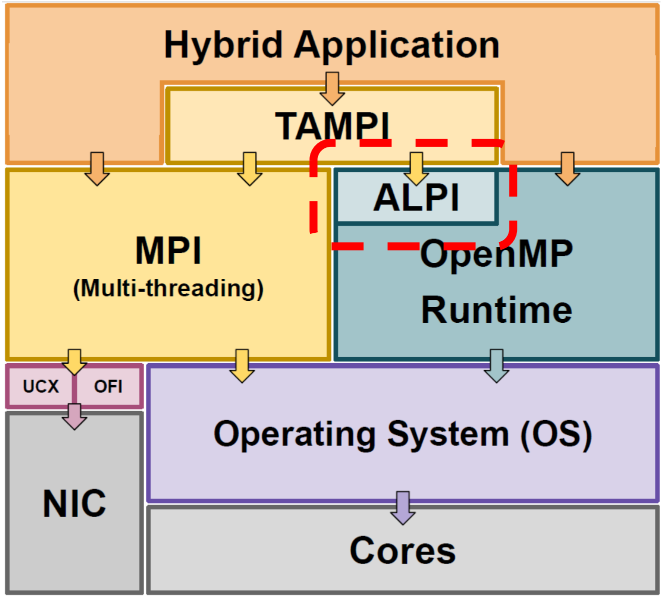
Outline

- Motivation
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
  - Hybrid MPI + OpenMP Programming
  - Task-Aware MPI Library
  - Gauss-Seidel Example
  - **Implementation**
- Task-Aware CUDA (TACUDA)
- Portability and Interoperability of TA-X Libraries

ALPI Interface

Implemented by the **tasking runtime system**!

```
// Get a handler of the calling task
int alpi_task_self(struct alpi_task **task);

// Pause and resume a task
int alpi_task_block(struct alpi_task *task);
int alpi_task_unblock(struct alpi_task *task);

// Register and fulfill events of a task
int alpi_task_events_increase(struct alpi_task *task, uint64_t increment);
int alpi_task_events_decrease(struct alpi_task *task, uint64_t decrement);

// Launch polling tasks
int alpi_task_spawn(void (*body)(void *), void *body_args,
               void (*completion_callback)(void *), void *completion_args,
               const char *label, const struct alpi_attr *attr);

// Pause the task for some time (e.g., polling task)
int alpi_task_waitfor_ns(uint64_t target_ns, uint64_t *actual_ns);
```



Full ALPI definition: https://gitlab.bsc.es/alpi/alpi

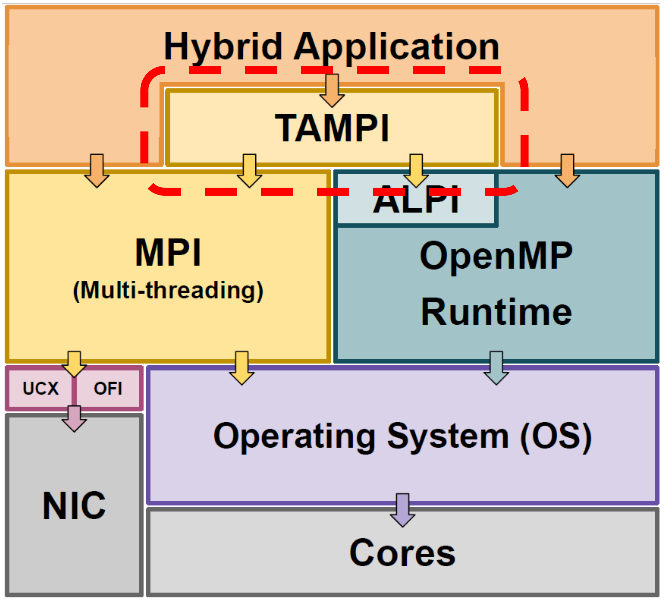
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **blocking** MPI operations

```
#pragma omp task depend(in: senddata[i]) depend(out: recvdata[i])
{
  MPI_Send(&senddata[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(&recvdata[i], 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  printf(”%d”, recvdata[i]);
}
```

```
int MPI_Recv(void *buffer, ..., MPI_Status *status) {
  int completed;
  MPI_Request request;
  MPI_Irecv(buffer, ..., &request);
  MPI_Test(&request, &completed, status);
  if (!completed) {
    alpi_task_t task = alpi_task_self();
    Ticket ticket(&request, status, task, /* blocking */ true);
    ticketQueue.push(ticket);
    alpi_task_block(task);
  }
}
```



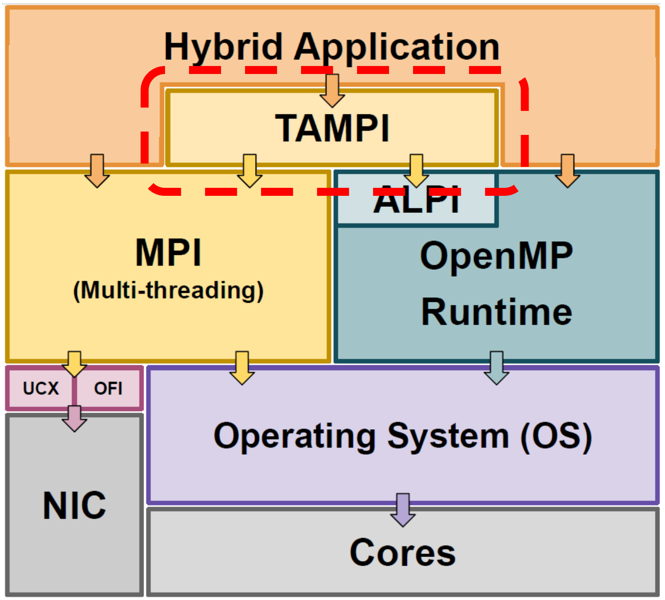
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **blocking** MPI operations

```
#pragma omp task depend(in: senddata[i]) depend(out: recvdata[i])
{
  MPI_Send(&senddata[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(&recvdata[i], 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  printf(”%d”, recvdata[i]);
}
```

```
int MPI_Recv(void *buffer, ..., MPI_Status *status) {
  int completed;
  MPI_Request request;
  MPI_Irecv(buffer, ..., &request);
  MPI_Test(&request, &completed, status);
  if (!completed) {
    alpi_task_t task = alpi_task_self();
    Ticket ticket(&request, status, task, /* blocking */ true);
    ticketQueue.push(ticket);
    alpi_task_block(task);
  }
}
```



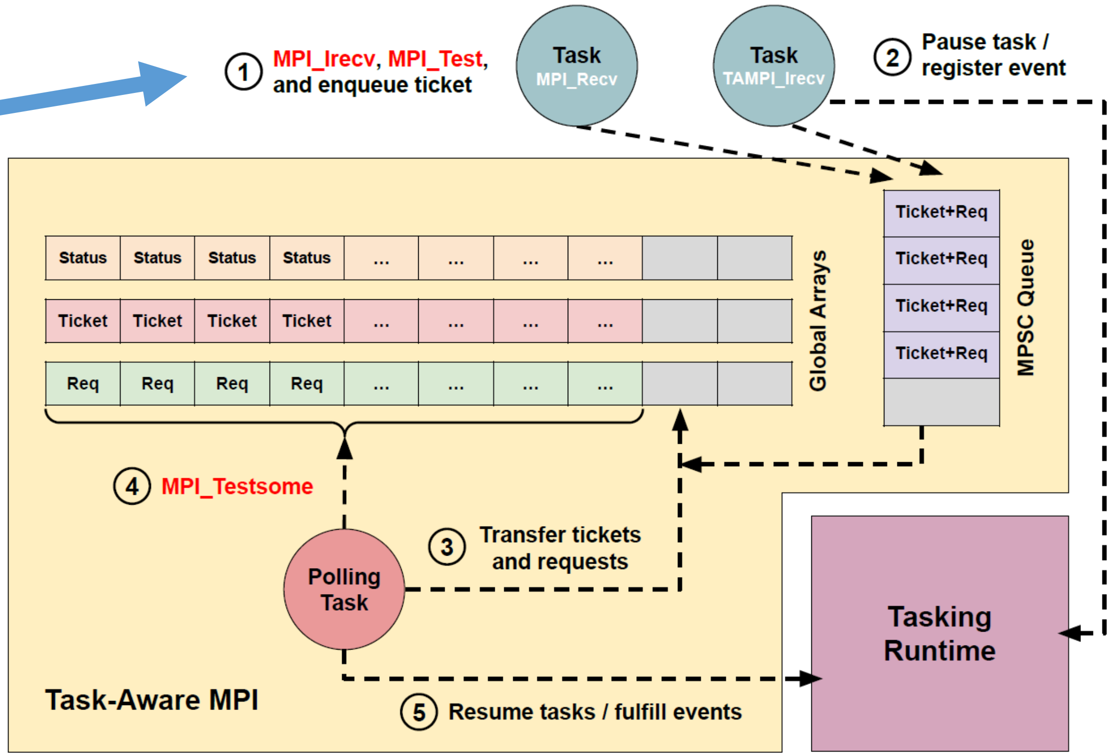
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **blocking** MPI operations

```
#pragma omp task depend(in: senddata[i]) depend(out: recvdata[i])
{
  MPI_Send(&senddata[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(&recvdata[i], 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  printf(”%d”, recvdata[i]);
}
```

```
int MPI_Recv(void *buffer, ..., MPI_Status *status) {
  int completed;
  MPI_Request request;
  MPI_Irecv(buffer, ..., &request);
  MPI_Test(&request, &completed, status);
  if (!completed) {
    alpi_task_t task = alpi_task_self();
    Ticket ticket(&request, status, task, /* blocking */ true);
    ticketQueue.push(ticket);
    alpi_task_block(task);
  }
}
```

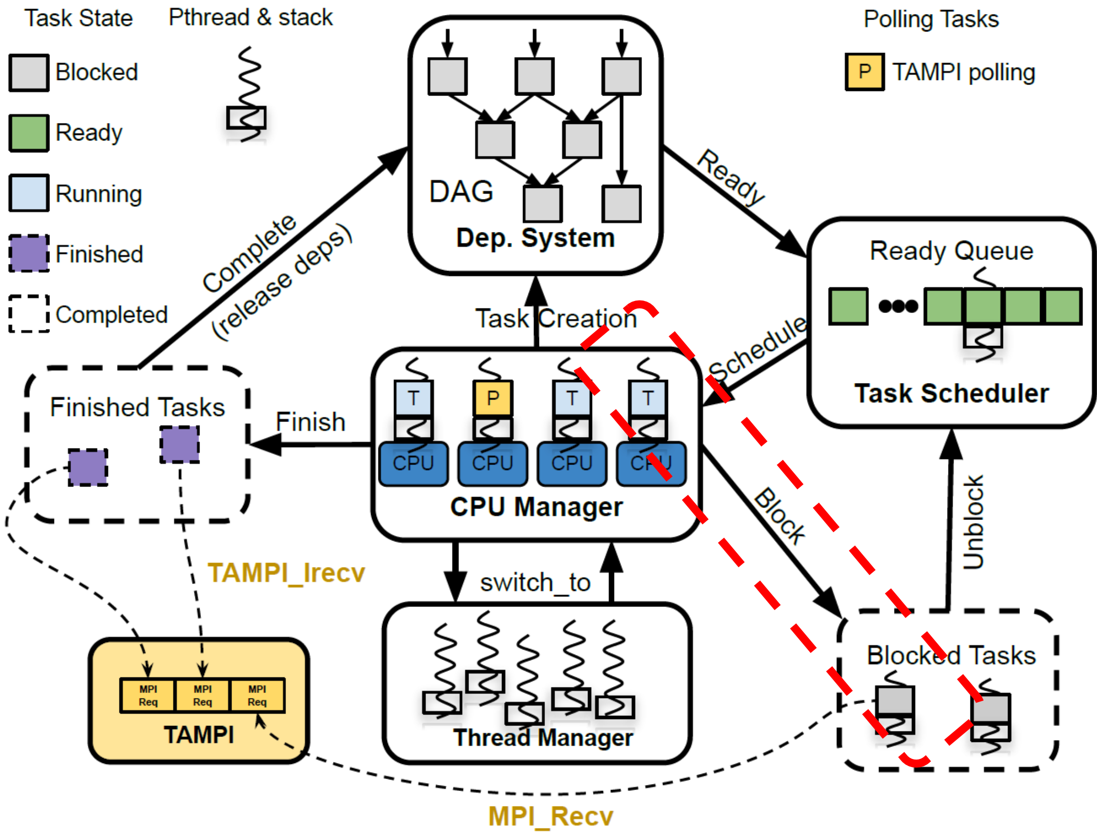


TAMPI Architecture and Implementation

Support for **blocking** MPI operations

```
#pragma omp task depend(in: senddata[i]) depend(out: recvdata[i])
{
  MPI_Send(&senddata[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  MPI_Recv(&recvdata[i], 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  printf("%d", recvdata[i]);
}
```

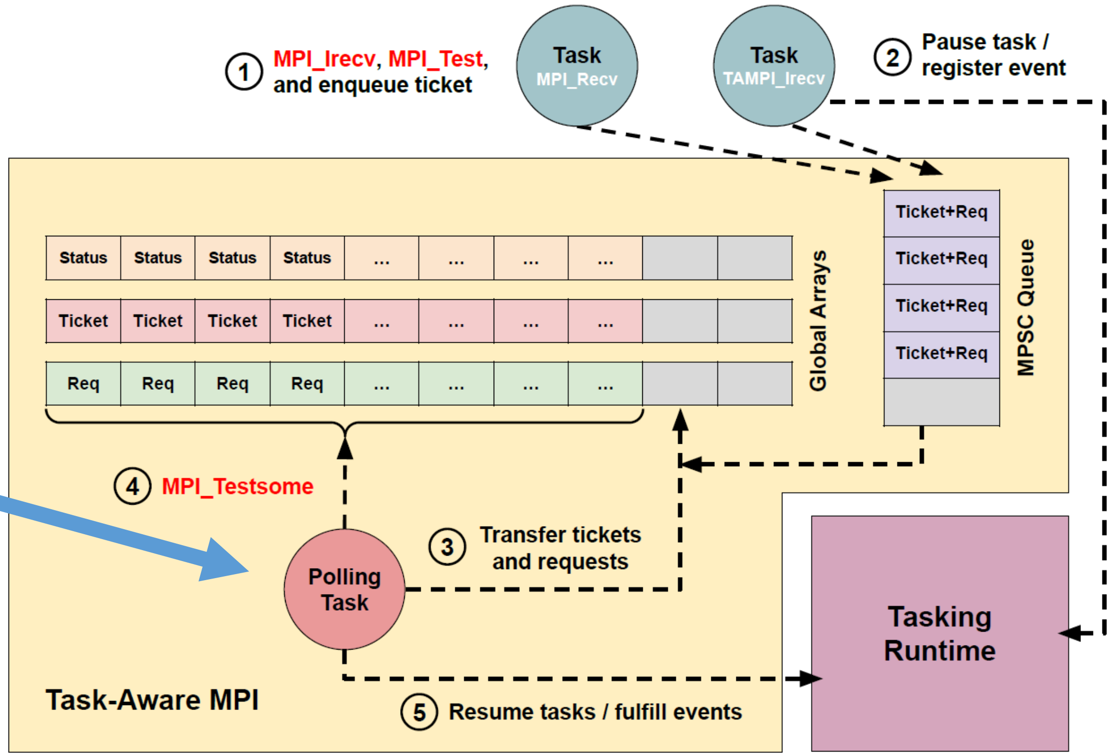
```
int MPI_Recv(void *buffer, ..., MPI_Status *status) {
  int completed;
  MPI_Request request;
  MPI_Irecv(buffer, ..., &request);
  MPI_Test(&request, &completed, status);
  if (!completed) {
    alpi_task_t task = alpi_task_self();
    Ticket ticket(&request, status, task, /* blocking */ false);
    ticketQueue.push(ticket);
    alpi_task_block(task);
  }
}
```



TAMPI Architecture and Implementation

Support for **blocking** MPI operations

```
void tampi::polling() {
  while (!tampi::shutdown) {
    vector<Ticket> tmpTickets = ticketQueue.pop();
    for (ticket : tmpTickets)
      globalArrays.add(ticket.request, ticket);

    MPI_Testsome(globalArrays.getRequests(), ...);
    vector<Ticket> completedTickets =
                  globalArrays.getTickets(...);

    for (ticket : completedTickets)
      if (ticket.blocking)
        alpi_task_unblock(ticket.task);
      else
        alpi_task_events_decrease(ticket.task, 1);

    alpi_task_waitfor_ns(/* polling period */ ...);
  }
}
```

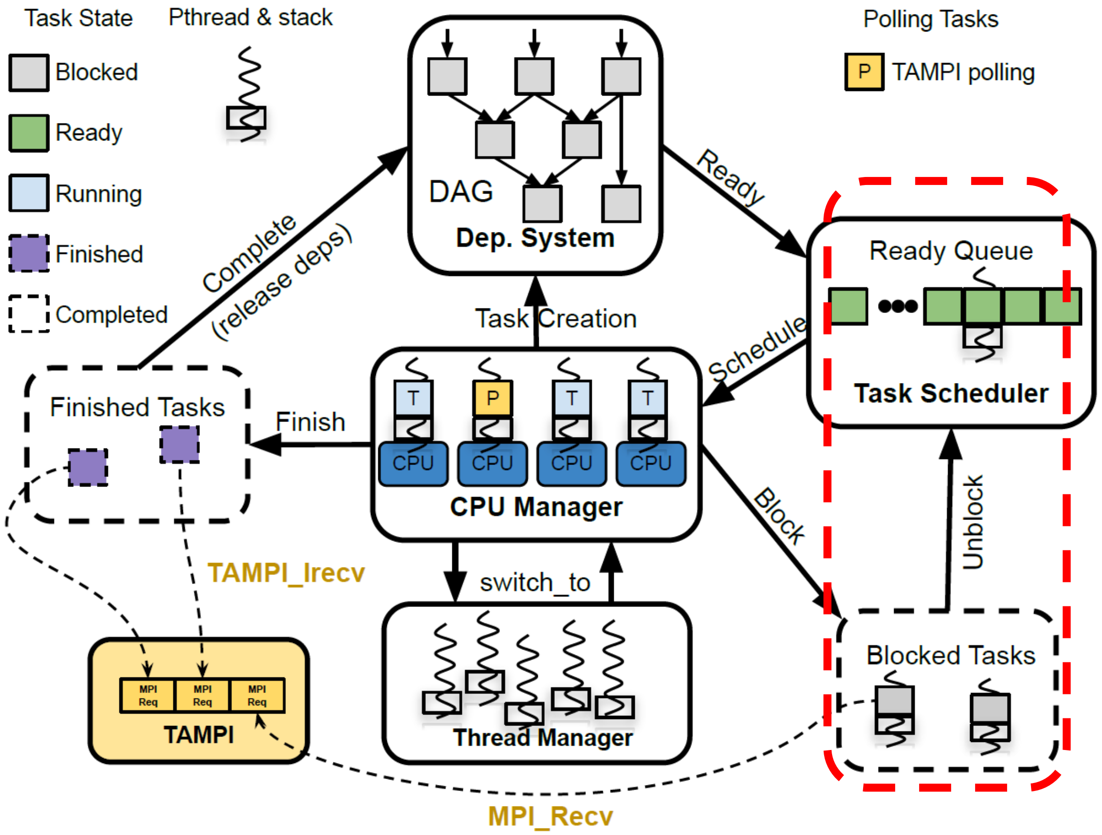


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **blocking** MPI operations

```
void tampi::polling() {
  while (!tampi::shutdown) {
    vector<Ticket> tmpTickets = ticketQueue.pop();
    for (ticket : tmpTickets)
      globalArrays.add(ticket.request, ticket);

    MPI_Testsome(globalArrays.getRequests(), ...);
    vector<Ticket> completedTickets =
                        globalArrays.getTickets(...);

    for (ticket : completedTickets)
      if (ticket.blocking)
        alpi_task_unblock(ticket.task);
      else
        alpi_task_events_decrease(ticket.task, 1);

    alpi_task_waitfor_ns(/* polling period */ ...);
  }
}
```



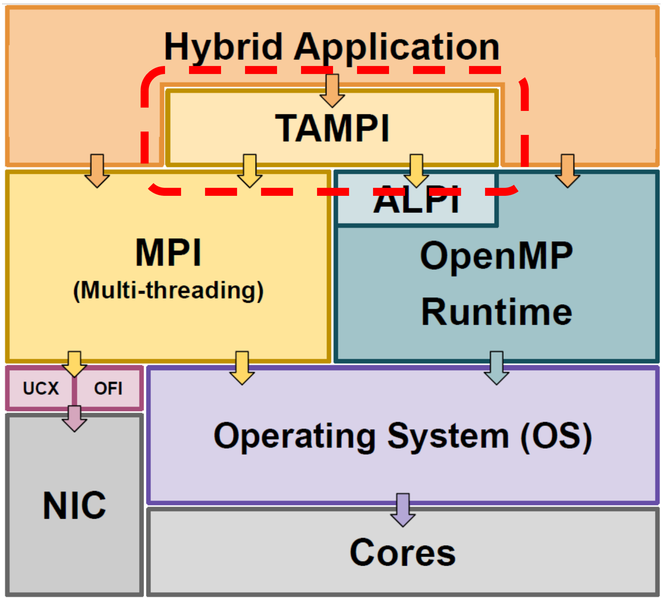
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **non-blocking** MPI operations

```
#pragma omp task depend(in: senddata[i]) depend(out: recvdata[i])
{
  TAMPI_Isend(&senddata[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  TAMPI_Irecv(&recvdata[i], 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  /* the operations are non-blocking: the buffers cannot be
     consumed or reused until the task has completed */
}
```

```
int TAMPI_Irecv(void *buffer, ..., MPI_Status *status) {
  int completed;
  MPI_Request request;
  MPI_Irecv(buffer, ..., &request);
  MPI_Test(&request, &completed, status);
  if (!completed) {
    alpi_task_t task = alpi_task_self();
    Ticket ticket(&request, status, task, /* nonblk */ false);
    alpi_task_events_increase(task, 1);
    ticketQueue.push(ticket);
  }
}
```



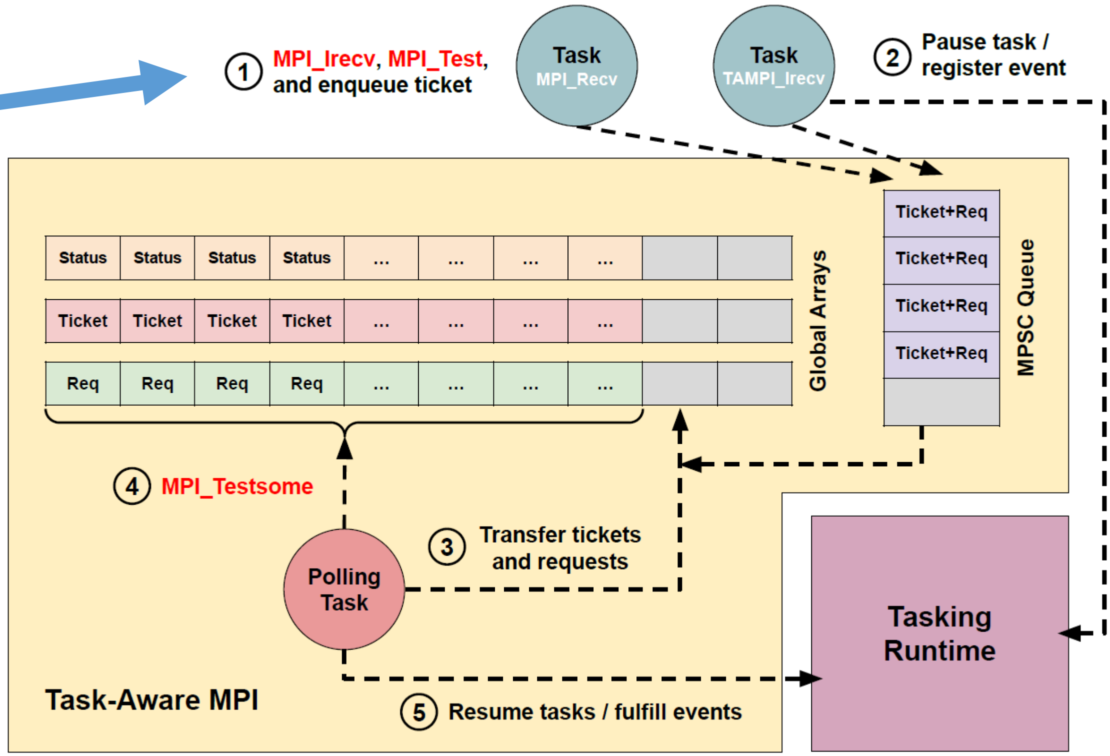
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **non-blocking** MPI operations

```
#pragma omp task depend(in: senddata[i]) depend(out: recvdata[i])
{
  TAMPI_Isend(&senddata[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  TAMPI_Irecv(&recvdata[i], 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  /* the operations are non-blocking: the buffers cannot be
     consumed or reused until the task has completed */
}
```

```
int TAMPI_Irecv(void *buffer, ..., MPI_Status *status) {
  int completed;
  MPI_Request request;
  MPI_Irecv(buffer, ..., &request);
  MPI_Test(&request, &completed, status);
  if (!completed) {
    alpi_task_t task = alpi_task_self();
    Ticket ticket(&request, status, task, /* nonblk */ false);
    alpi_task_events_increase(task, 1);
    ticketQueue.push(ticket);
  }
}
```



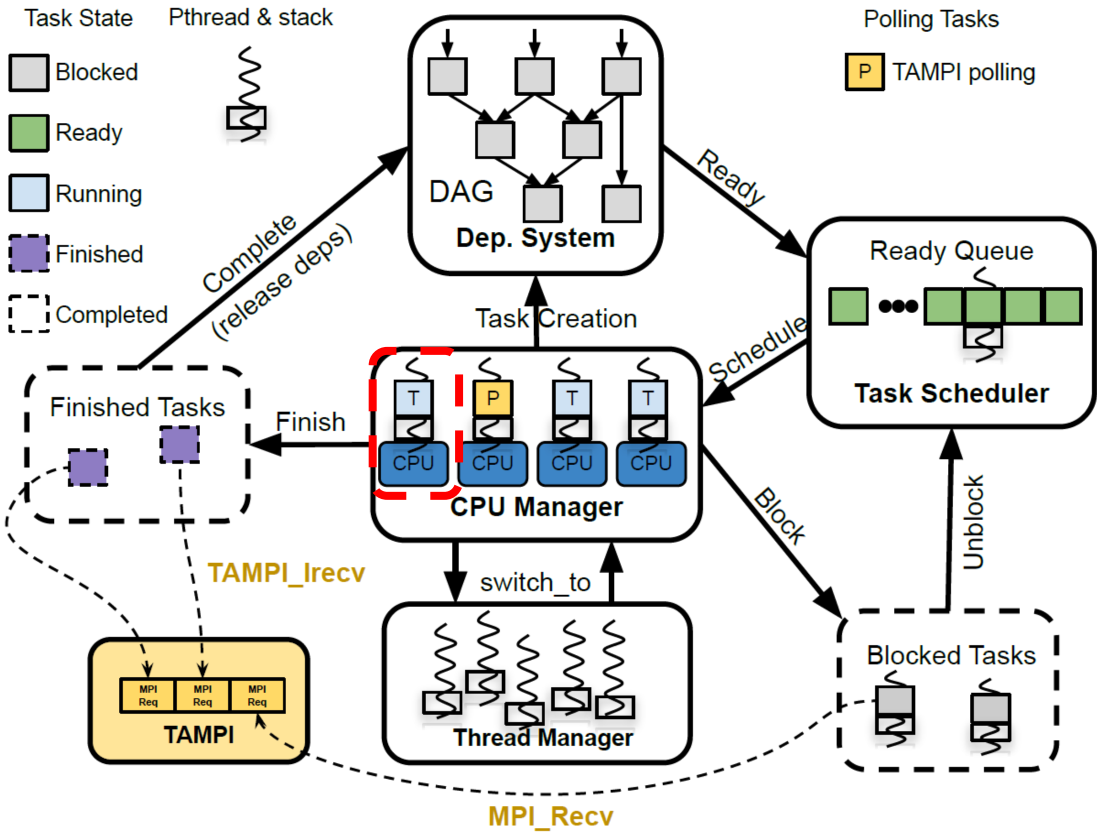
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **non-blocking** MPI operations

```
#pragma omp task depend(in: senddata[i]) depend(out: recvdata[i])
{
  TAMPI_Isend(&senddata[i], 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
  TAMPI_Irecv(&recvdata[i], 1, MPI_INT, 0, tag, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  /* the operations are non-blocking: the buffers cannot be
     consumed or reused until the task has completed */
}
```

```
int TAMPI_Irecv(void *buffer, ..., MPI_Status *status) {
  int completed;
  MPI_Request request;
  MPI_Irecv(buffer, ..., &request);
  MPI_Test(&request, &completed, status);
  if (!completed) {
    alpi_task_t task = alpi_task_self();
    Ticket ticket(&request, status, task, /* nonblk */ false);
    alpi_task_events_increase(task, 1);
    ticketQueue.push(ticket);
  }
}
```

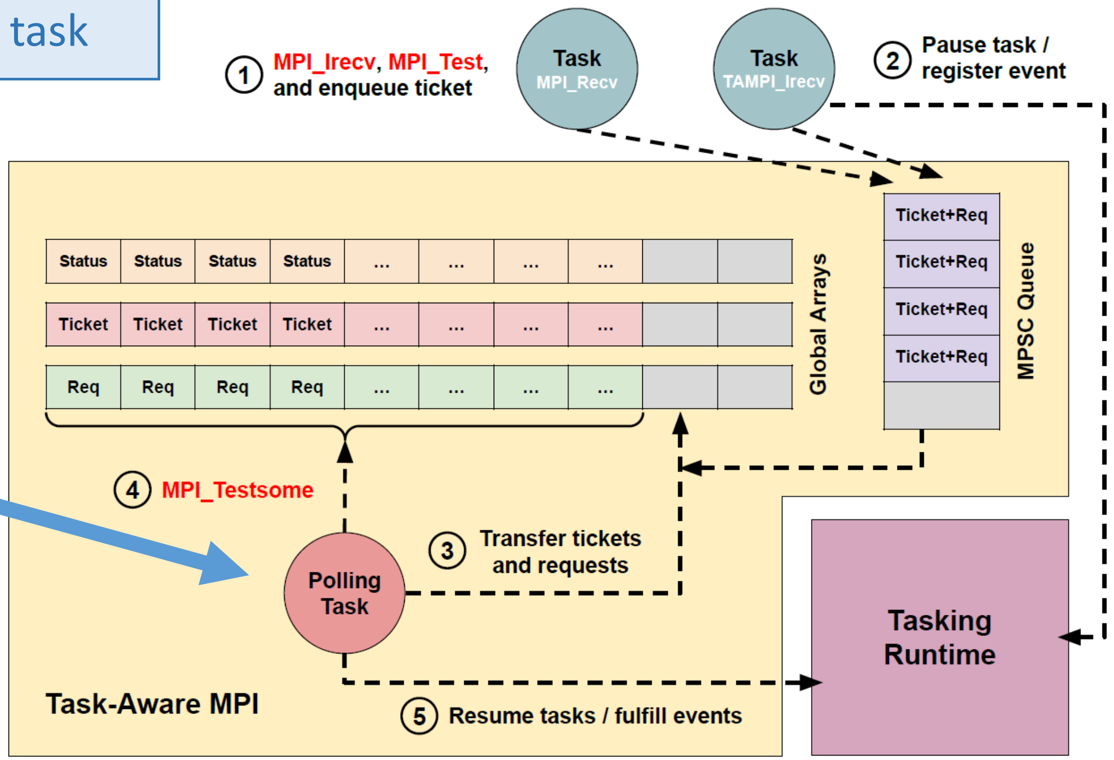


TAMPI Architecture and Implementation

Support for **non-blocking** MPI operations

Same polling task

```
void tampi::polling() {
  while (!tampi::shutdown) {
    vector<Ticket> tmpTickets = ticketQueue.pop();
    for (ticket : tmpTickets)
      globalArrays.add(ticket.request, ticket);

    MPI_Testsome(globalArrays.getRequests(), ...);
    vector<Ticket> completedTickets =
                         globalArrays.getTickets(...);

    for (ticket : completedTickets)
      if (ticket.blocking)
        alpi_task_unblock(ticket.task);
      else
        alpi_task_events_decrease(ticket.task, 1);

    alpi_task_waitfor_ns(/* polling period */ ...);
  }
}
```

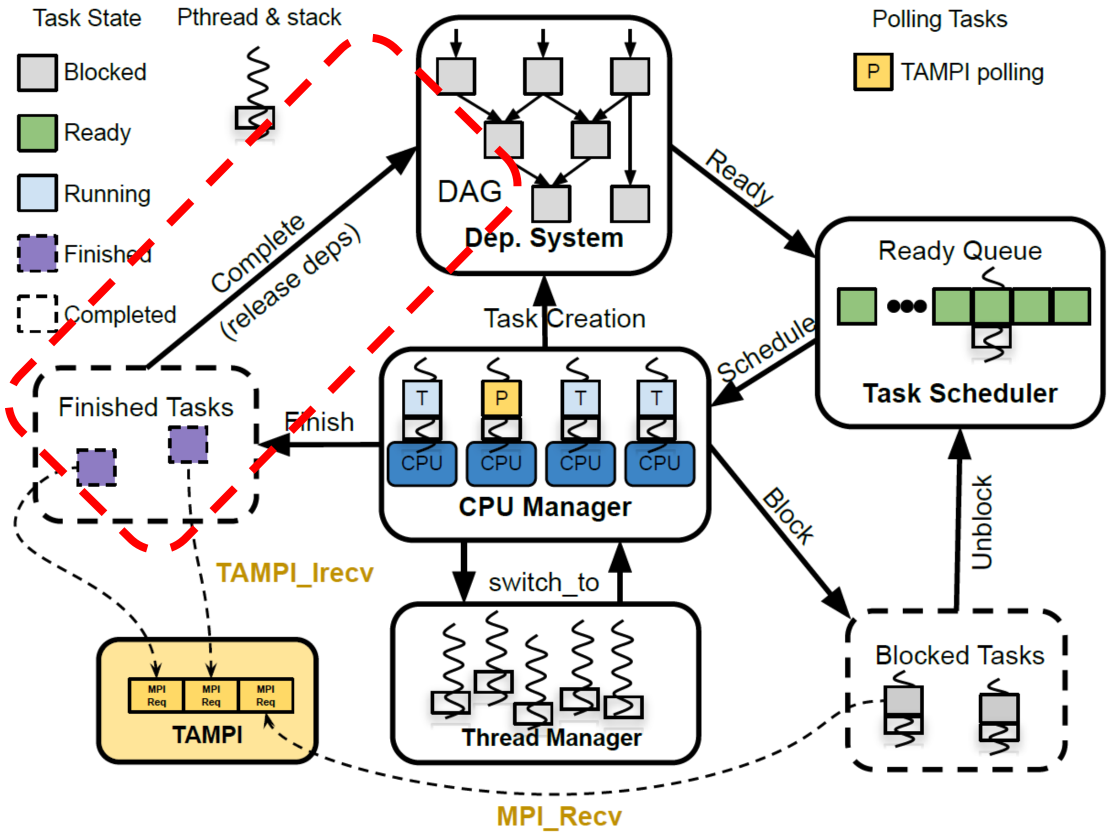


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Support for **non-blocking** MPI operations

```
void tampi::polling() {
  while (!tampi::shutdown) {
    vector<Ticket> tmpTickets = ticketQueue.pop();
    for (ticket : tmpTickets)
      globalArrays.add(ticket.request, ticket);

    MPI_Testsome(globalArrays.getRequests(), ...);
    vector<Ticket> completedTickets =
                        globalArrays.getTickets(...);

    for (ticket : completedTickets)
      if (ticket.blocking)
        alpi_task_unblock(ticket.task);
      else
        alpi_task_events_decrease(ticket.task, 1);

    alpi_task_waitfor_ns(/* polling period */ ...);
  }
}
```

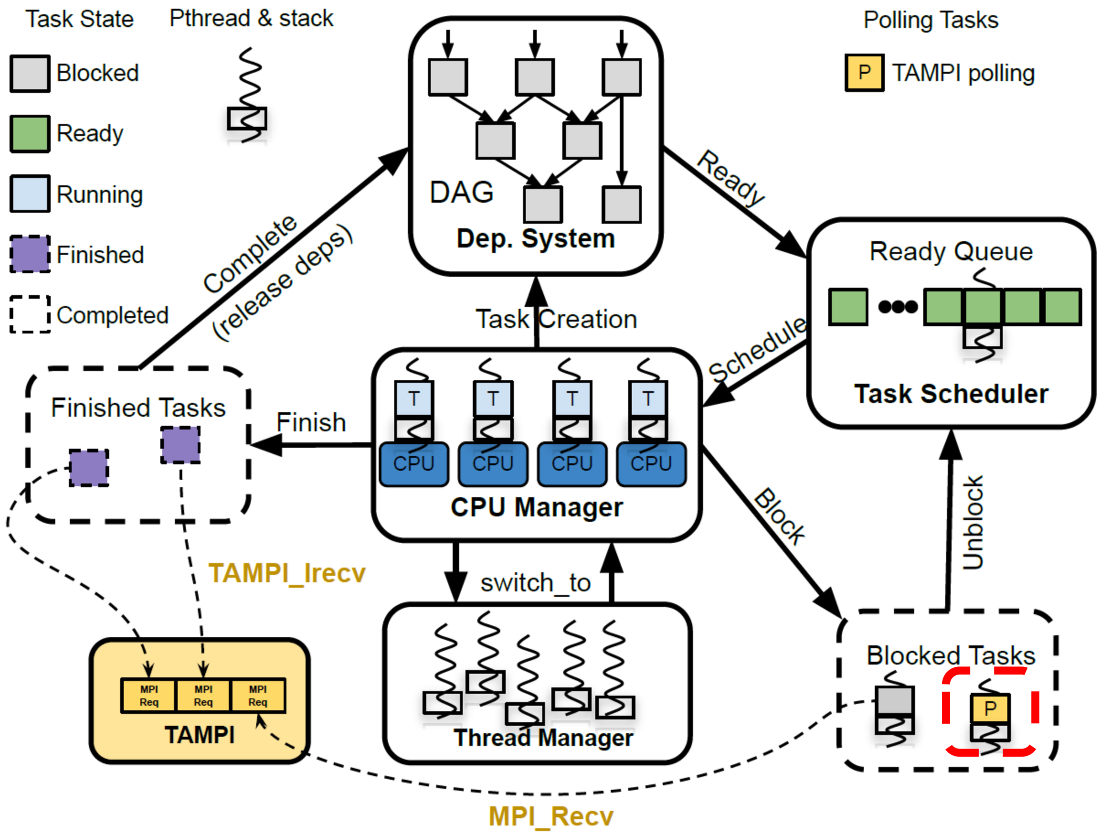


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TAMPI Architecture and Implementation

Efficient **asynchrounous polling** and **progress**

```
void tampi::polling() {
  while (!tampi::shutdown) {
    vector<Ticket> tmpTickets = ticketQueue.pop();
    for (ticket : tmpTickets)
      globalArrays.add(ticket.request, ticket);

    MPI_Testsome(globalArrays.getRequests(), ...);
    vector<Ticket> completedTickets =
                        globalArrays.getTickets(...);

    for (ticket : completedTickets)
      if (ticket.blocking)
        alpi_task_unblock(ticket.task);
      else
        alpi_task_events_decrease(ticket.task, 1);

    alpi_task_waitfor_ns(/* polling period */ ...);
  }
}
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-Aware CUDA

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Outline

- Motivation
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- **Task-Aware CUDA (TACUDA)**
  - Hybrid CUDA + OpenMP Programming
  - Task-Aware CUDA (TACUDA)
  - Cholesky Example
  - Implementation
- Portability and Interoperability of TA-X Libraries

Programming Heterogeneous Systems

Heterogeneity at the node level and …

- Multi-cores, GPUs, FPGAs, AI accelerators, SmartNICs

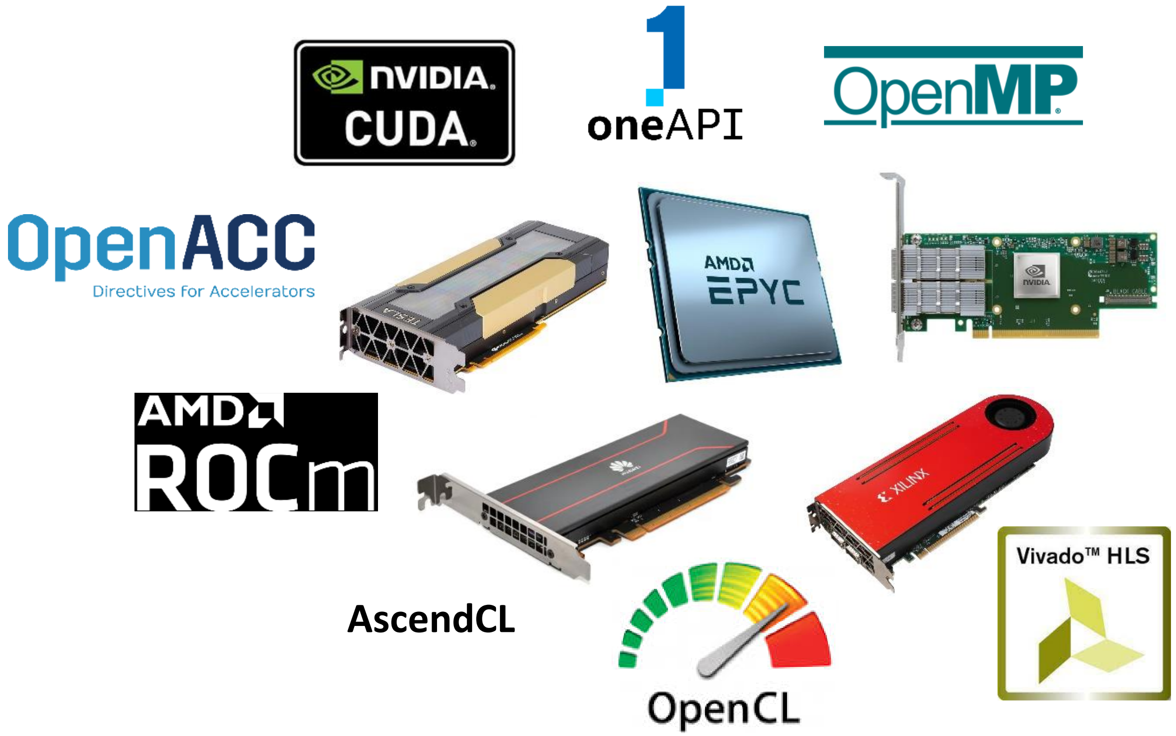
Heterogeneity at the device level

- Multi-cores with on-chip accelerators (compression, encryption, etc)
- GPUs with tensor cores
- FPGAs with custom DSPs/processors

Plethora of programming models and APIs

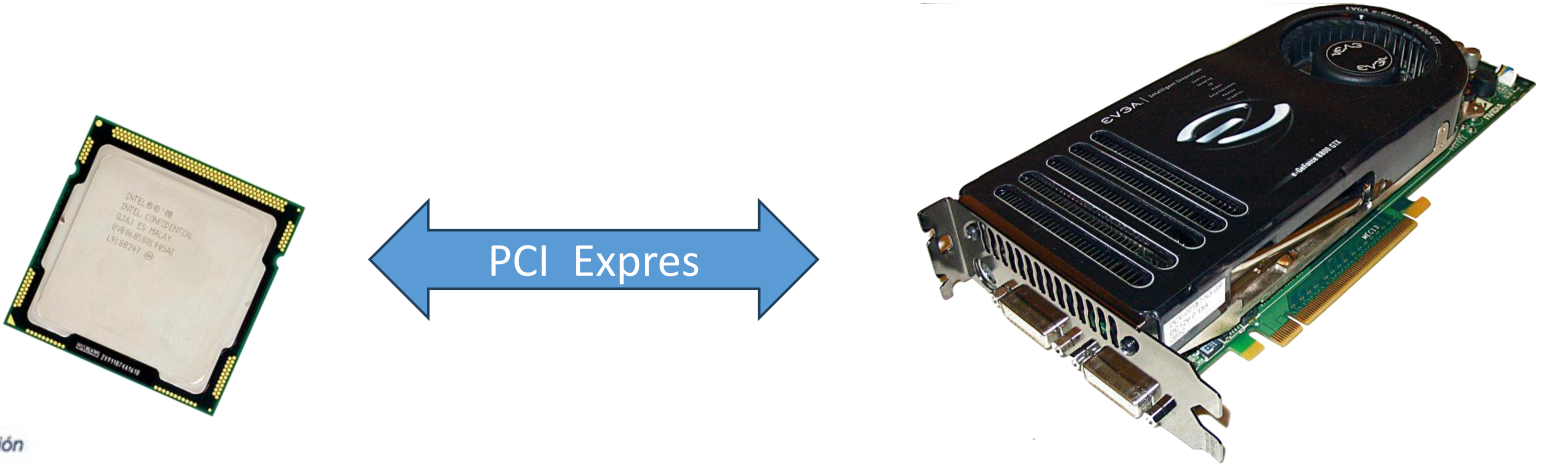
- Each vendors has its own API
- Several “standards”

Still have to be combined with host programming model (pthreads, OpenMP, etc) and **MPI**!



First GPU-based accelerators

- Intel Xeon (4 cores) vs NVIDIA Geoforce 8 (2008)
- PCI Express is a bottleneck
  - High latency
  - Low Bandwidth
- Recommended use
  - Host: GPU management and communication with other hosts
  - GPU: main computation (big fat CUDA kernel)
- Fine-grained cooperation between CPU and GPU **unfeasible**



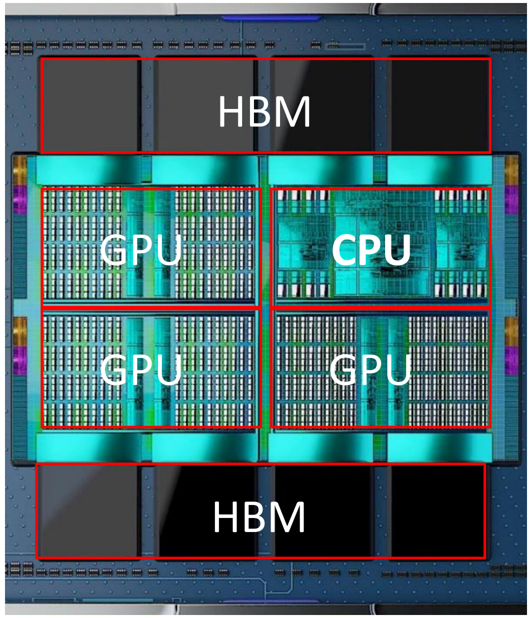
Software Evolution (CUDA)

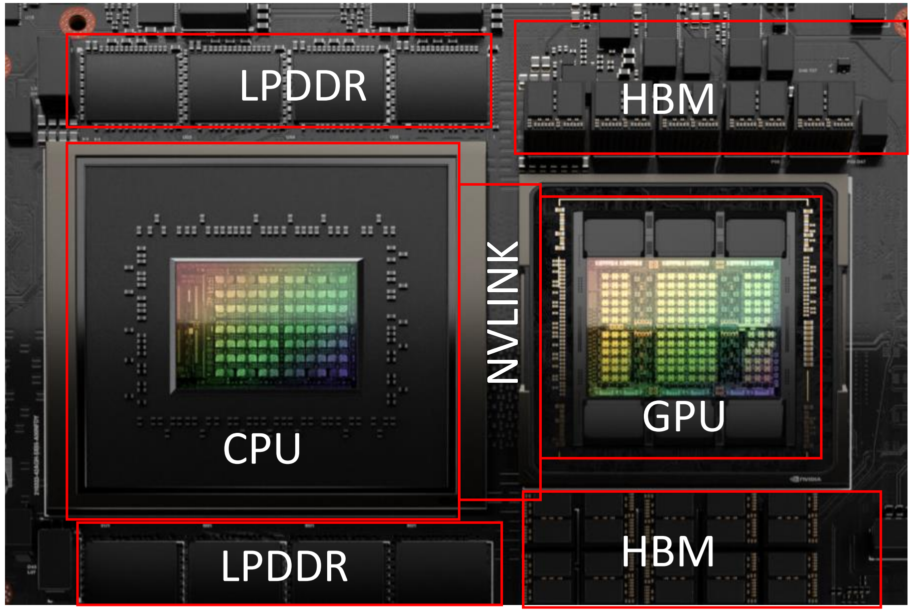
- CUDA 1.0: **cudaMemcpy**
- CUDA 2.0: cudaHostAlloc(…, **cudaHostAllocMapped**): Direct (but slow) access to host memory
- CUDA 4.0: **Unified Virtual Memory** (same virtual address space can be used on host and device)
- CUDA 5.0: **Dynamic Parallelism**
- CUDA 6.0: **Unified Memory** (software-based) (host and device can work on the same memory)
- CUDA 7.0: **cudaMemcpyAsync**
- CUDA 8.0: **Unified Memory** (hardware-based) (host and device can work on the same memory)
- CUDA 9.0: **MPS**
- CUDA 10: **CUDA Graphs**
- CUDA 12: **New CUDA Dynamic Parallelism APIs**

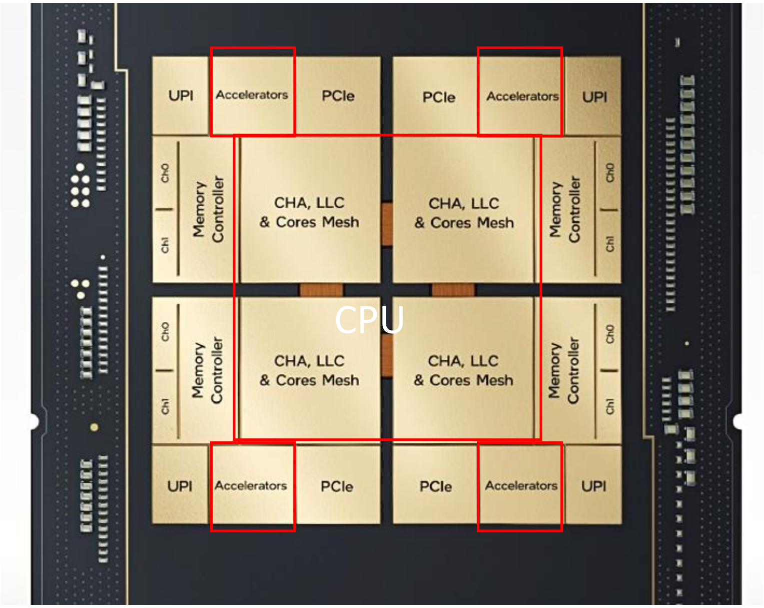
Programming Heterogeneous Systems

AMD MI300 fully shared identical memory vs NVIDIA Grace Hopper fully coherent discrete memory

Intel Sapphire Rapids: coherent shared memory space, across CPU cores and DSA and QAT acceleration engines







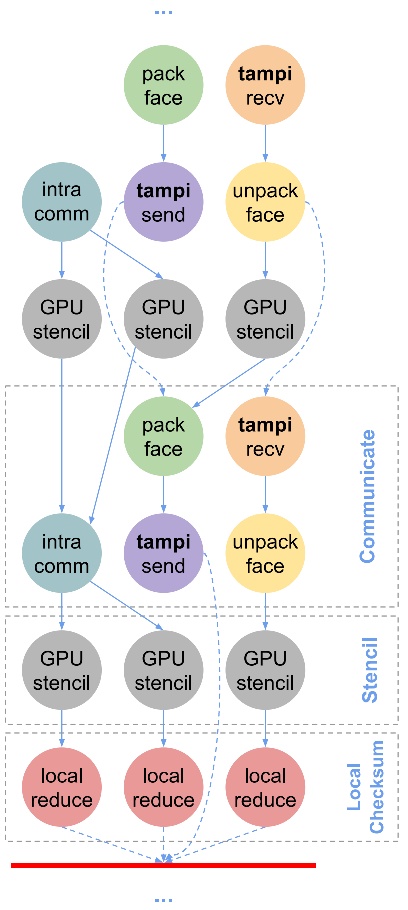
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

How OpenMP fit in this heterogeneous world?

OpenMP dataflow model to **orchestrate** computations, data transfers and communications

Leverage and interoperate with other languages and APIs to exploit accelerators

- **CUDA C**, OpenCL C, pragma omp simd, OpenACC pragmas, Xilinx HLS, etc.
- Optimized libraries: **cuBLAS**, **cuFFT**, mkl, etc
- MPI and GASPI (distributed systems)



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CUDA Overview

CUDA API

- Set of APIs to manage GPU devices and orchestrate host and device data transfers and computations

CUDA C

- Programming language similar to C/C++ specially designed to develop kernels that can exploit GPU architectures

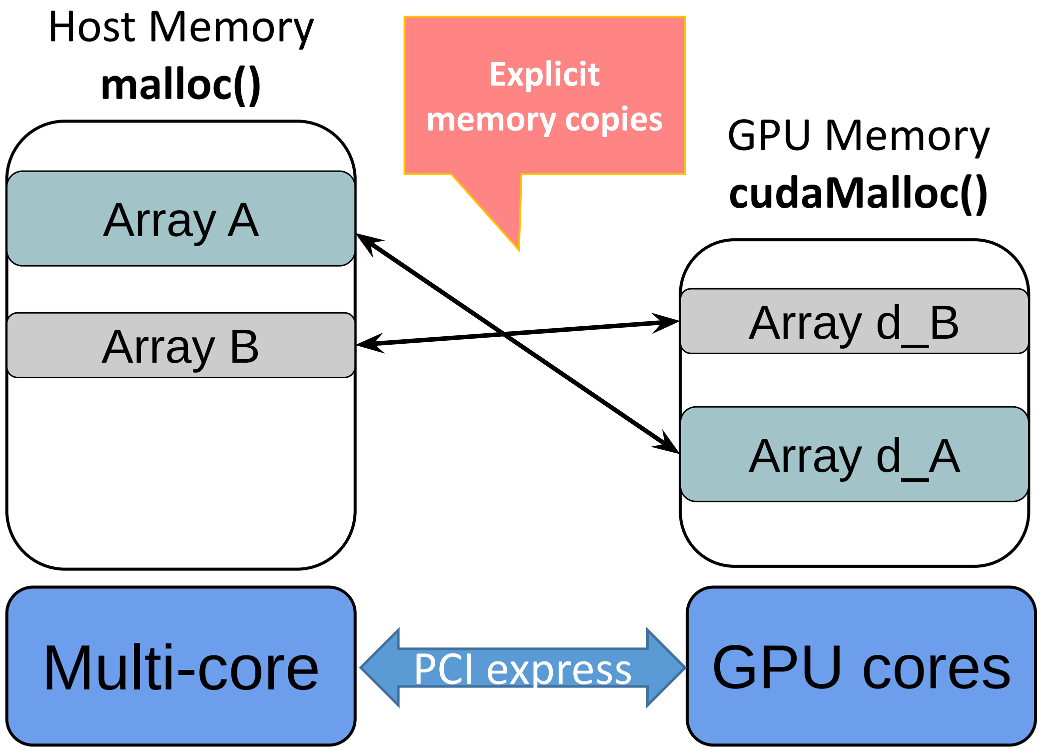
Memory management

- Transparent (Unified Memory)
- **Manual (Explicit copies)**

NVIDIA CUDA

CUDA Overview

Explicit Memory Management



CUDA Hello World (Synchronous)

```
int main(void)
{
  /* … */
  // Allocate host memory
  x = malloc(N*sizeof(float));
  y = malloc(N*sizeof(float));

  // Allocate device memory
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, sizeof(float) * N,
             cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, sizeof(float) * N,
             cudaMemcpyHostToDevice);
```

Host Allocation

Device Allocation

Explicit Copies

```
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1)/blockSize;

  // Async kernel launch
  add<<<numBlocks, blockSize>>>(N, d_x, d_y);

  // Async kernel launch
  cudaDeviceSynchronize();

  cudaMemcpy(y, d_y, sizeof(float) * N,
             cudaMemcpyDeviceToHost);

  // Free memory
  free(x); cudaFree(d_x);
  free(y); cudaFree(d_y);
  return 0;
}
```

Kernel Launch

Host/Device synchronization

Explicit Copies

```
// Kernel function to add elements of two arrays
__global__ void add(int n, float *x, float *y) {
  for (int i = threadIdx.x; i < n; i += blockDim.x)
      y[i] = x[i] + y[i];
}
```

CUDA Kernel

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

CUDA Hello World (Async)

```
int main(void)
{
  // Allocate host & device buffers
  ...

  cudaStream_t stream;
  cudaStreamCreate(&stream);

  cudaMemcpyAsync(d_x, x, sizeof(float) * N,
                  cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_y, y, sizeof(float) * N,
                  cudaMemcpyHostToDevice, stream);

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1)/blockSize;

  // Async kernel launch
  add<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
```

Stream creation

Async Copies

Async Kernel Launch

```
  cudaMemcpyAsync(y, d_y, sizeof(float) * N,
                  cudaMemcpyDeviceToHost, stream);

  // Sync with all ops on the stream
  cudaStreamSynchronize(stream);

  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

Async Copies

Stream Synchronization

CUDA Hello World (Async copies) + Tasks

```
int main(void)
{
  /* Alloc host & device buffers */
  #pragma omp task depend(in: x[0;N]) \
          depend(inout: y[0;N], d_y[0;N], d_x[0;N])
  {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    cudaMemcpyAsync(d_x, x, sizeof(float) * N,
                    cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(d_y, y, sizeof(float) * N,
                    cudaMemcpyHostToDevice, stream);

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1)/blockSize;

    // Async kernel launch
    add<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
```

```
    cudaMemcpyAsync(y, d_y, sizeof(float) * N,
                    cudaMemcpyDeviceToHost, stream);

    // Sync with all ops on the stream
    cudaStreamSynchronize(stream);

  } // End of task

  #pragma omp taskwait

  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

Any problem with this code?

CUDA Hello World (Async copies) + Tasks

```
int main(void)
{
  /* Alloc host & device buffers */
  #pragma omp task depend(in: x[0;N]) \
           depend(inout: y[0;N], d_y[0;N], d_x[0;N])
  {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    cudaMemcpyAsync(d_x, x, sizeof(float) * N,
                    cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(d_y, y, sizeof(float) * N,
                    cudaMemcpyHostToDevice, stream);

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1)/blockSize;

    // Async kernel launch
    add<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
```

```
    cudaMemcpyAsync(y, d_y, sizeof(float) * N,
                    cudaMemcpyDeviceToHost, stream);

    // Sync with all ops on the stream
    cudaStreamSynchronize(stream);

  } // End of task

  #pragma omp taskwait

  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

This call will block the task and the CPU!

Any problem with this code?

Outline

- Motivation
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
  - Hybrid CUDA + OpenMP Programming
  - **Task-Aware CUDA (TACUDA)**
  - Cholesky Example
  - Implementation
- Portability and Interoperability of TA-X Libraries

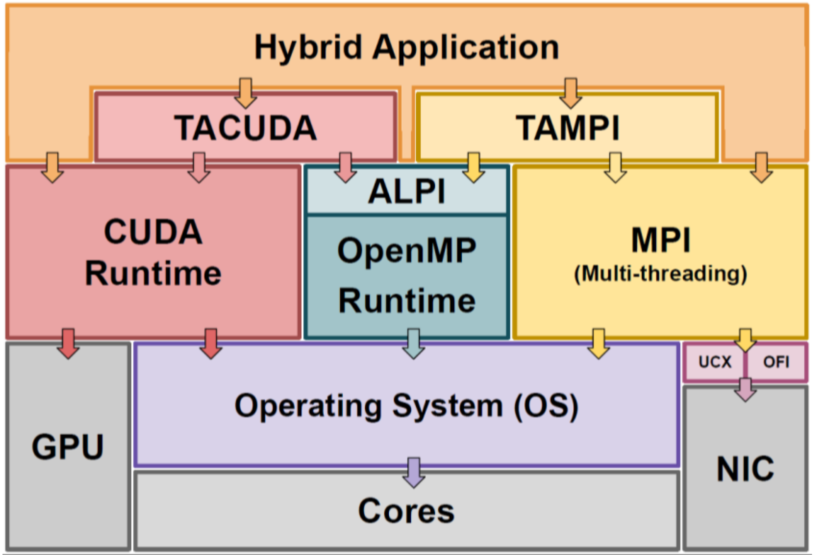
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

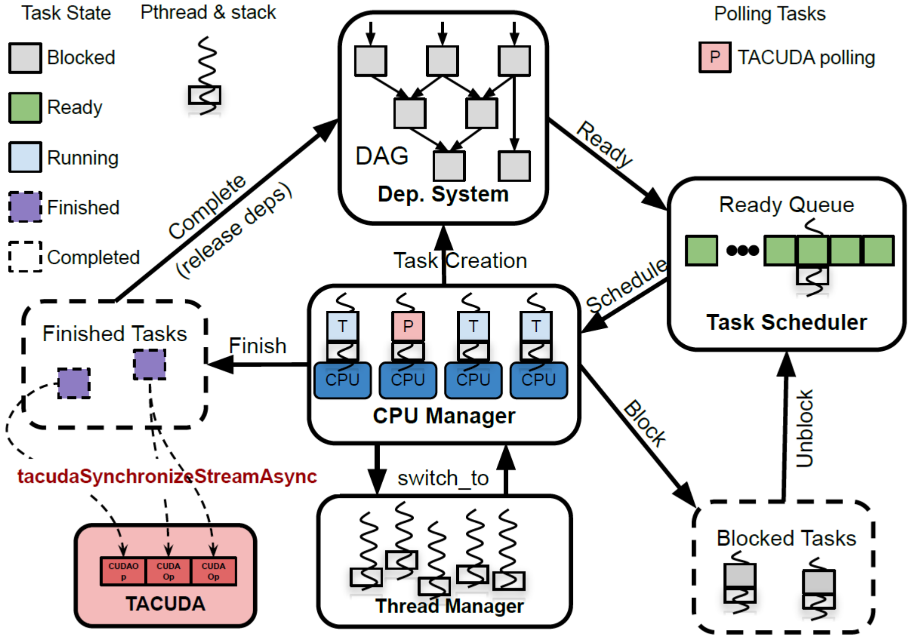
Task-Aware CUDA (TACUDA)

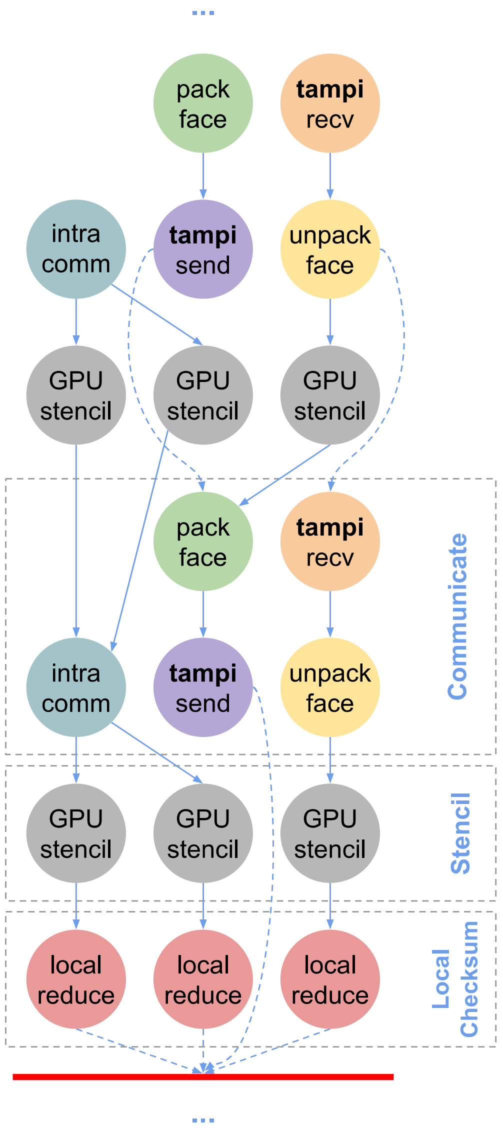
Independent **library** to develop OpenMP **heterogeneous** applications

- **Taskifying** computation, **data-transfers** and **offloading** phases
- Natural **overlap** of data. transfers and computations on CPU and GPU

TACUDA can be mixed with TAMPI to develop distributed heterogeneous applications!







Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-Aware CUDA (TACUDA) API

```
//! Initializes the TACUDA environment
CUresult tacudaInit(unsigned int flags);

//! Finalizes the TACUDA environment
CUresult tacudaFinalize();

//! Initializes the pool of stream
cudaError_t tacudaCreateStreams(size_t count);

//! Finalization of the pool of streams
cudaError_t tacudaDestroyStreams();

//!  Gets a stream from the pool
cudaError_t tacudaGetStream(cudaStream_t *stream);

//! \brief Returning a stream to the pool
cudaError_t tacudaReturnStream(cudaStream_t stream);
```

Pool of streams for tasks

https://github.com/bsc-pm/tacuda/-/blob/master/src/include/TACUDA.h

Task-Aware CUDA (TACUDA) API

```
//! Asynchronous function, binds the completion of the calling task to the finalization of the
submitted operations on the stream
cudaError_t tacudaSynchronizeStreamAsync(cudaStream_t stream);

//! Copying of data between host and device
__host__ __device__ cudaError_t tacudaMemcpyAsync(void *dst, const void *src, size_t sizeBytes,
                          enum cudaMemcpyKind kind, cudaStream_t stream, tacudaRequest *request);

//! \brief Launching a device function
__host__ cudaError_t tacudaLaunchKernel(const void* func, dim3 gridDim, dim3 blockDim,
                 void** args, size_t sharedMem, cudaStream_t stream, tacudaRequest *request);

//! Asynchronous and non-blocking operation, binds a request to the calling task
cudaError_t tacudaWaitRequestAsync(tacudaRequest *request);

//! Bindig multiple requests
cudaError_t tacudaWaitallRequestsAsync(size_t count, tacudaRequest requests[]);
```

Async wait of stream

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

https://github.com/bsc-pm/tacuda/-/blob/master/src/include/TACUDA.h

CUDA Hello World (Async copies)

```
int main(void)
{
  /* Alloc host & device buffers */

  cudaStream_t stream;
  cudaStreamCreate(&stream);


  cudaMemcpyAsync(d_x, x, sizeof(float) * N,
                 cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_y, y, sizeof(float) * N,
                 cudaMemcpyHostToDevice, stream);

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1)/blockSize;

  // Async kernel launch
  add<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
```

Stream creation

Async Copies

Kernel Launch

```
  cudaMemcpyAsync(y, d_y, sizeof(float) * N,
                 cudaMemcpyDeviceToHost, stream);

  // Sync with all ops on the stream
  cudaStreamSynchronize(stream);

  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

Async Copies

Stream synchronization

RECAP!

CUDA Hello World (Async copies) + Tasks

```
int main(void)
{
  /* Alloc host & device buffers */
  #pragma omp task depend(in: x[0;N]) \
           depend(inout: y[0;N], d_y[0;N], d_x[0;N])
  {
    cudaStream_t stream;
    cudaStreamCreate(&stream);

    cudaMemcpyAsync(d_x, x, sizeof(float) * N,
                    cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(d_y, y, sizeof(float) * N,
                    cudaMemcpyHostToDevice, stream);

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1)/blockSize;

    // Async kernel launch
    add<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
```

```
    cudaMemcpyAsync(y, d_y, sizeof(float) * N,
                    cudaMemcpyDeviceToHost, stream);

    // Sync with all ops on the stream
    cudaStreamSynchronize(stream);

  } // End of task

  #pragma omp taskwait

  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

This call will block the task and the CPU!

RECAP!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

TACUDA Hello World (Async copies)

```
int main(void)
{
  /* Alloc host & device buffers */
  #pragma omp task depend(in: x[0;N]) \
           depend(inout: y[0;N], d_y[0;N], d_x[0;N])
  {
    cudaStream_t stream;
    tacudaGetStream(&stream);

    cudaMemcpyAsync(d_x, x, sizeof(float) * N,
            cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync(d_y, y, sizeof(float) * N,
            cudaMemcpyHostToDevice, stream);

    int blockSize = 256;
    int numBlocks = (N + blockSize - 1)/blockSize;
    // Async kernel launch
    add<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
```

Get stream

Async Copies

Kernel Launch

```
    cudaMemcpyAsync(y, d_y, sizeof(float) * N,
             cudaMemcpyDeviceToHost, stream);

    // Sync with all ops on the stream
    tacudaStreamSynchronizeAsync(stream);
    tacudaReturnStream(stream);
  } // End of task

  #pragma omp taskwait

  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```

Async Copies

Bind stream operations

OpenMP synchronization

- By using TACUDA, the CPU that executes the task will not block
- OpenMP dependences or *taskwait* can be used as a synchronization method
- Notice that device buffers have to be annotated!

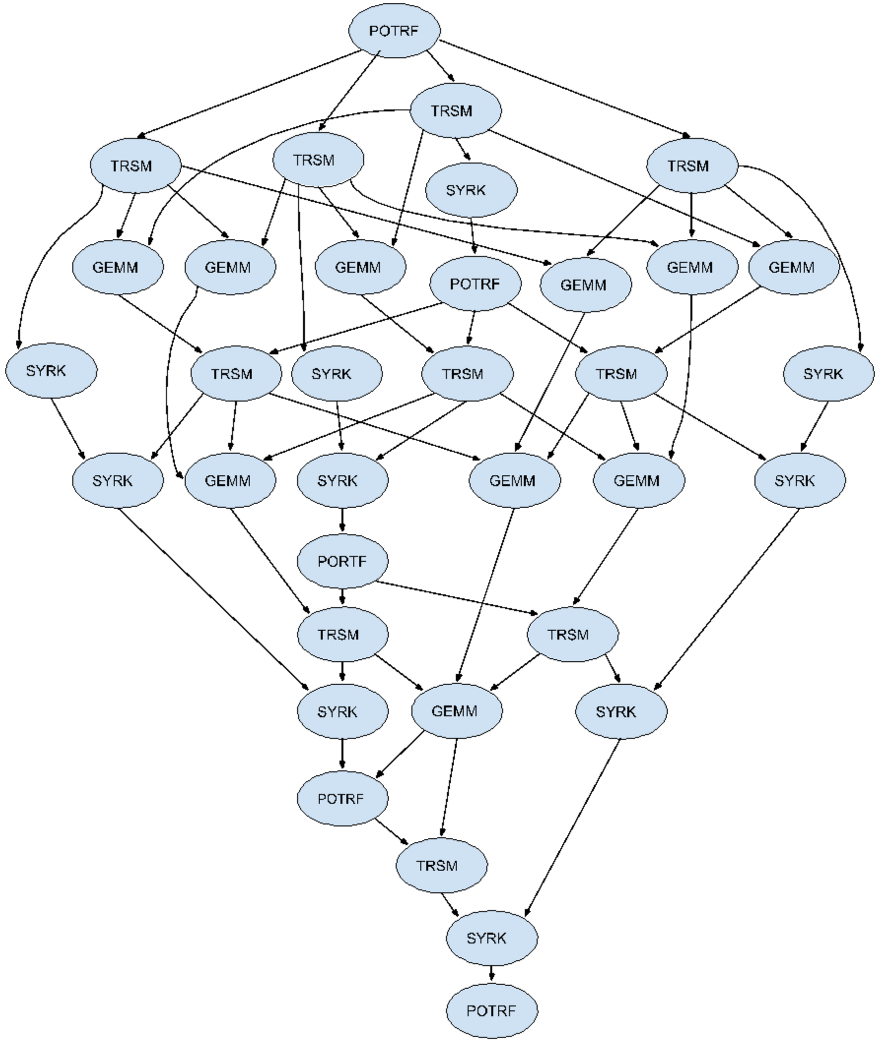
Outline

- Motivation
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
    - Hybrid CUDA + OpenMP Programming
    - Task-Aware CUDA (TACUDA)
    - **Cholesky Example**
    - Implementation
- Portability and Interoperability of TA-X Libraries

Cholesky factorization: OpenMP tasks

Cholesky factorization

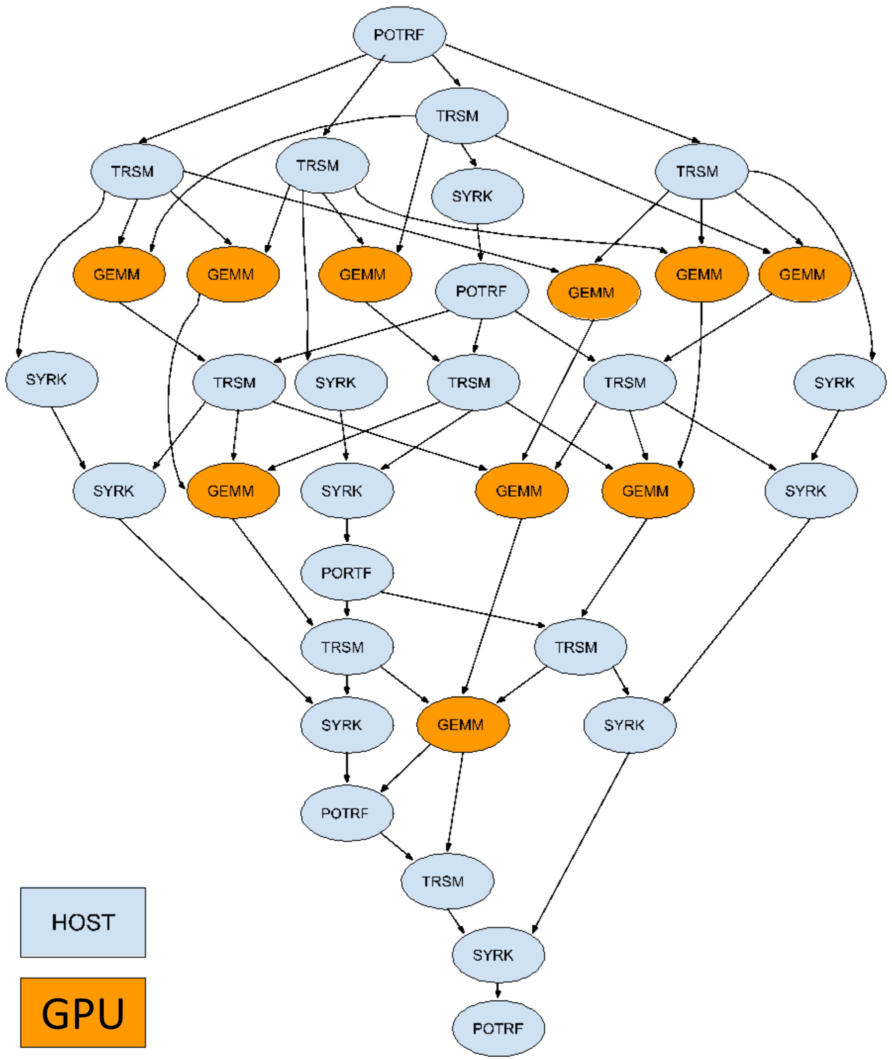
```
void cholesky(size_t N, size_t TS, double (*A)[N/TS][TS][TS]) {
  for (long k = 0; k < N/TS; k++) {
    #pragma omp task depend(inout: A[k][k])
    LAPACKE_dpotrf(..., A[k][k], TS);

    for (long i = k+1; i < N/TS; i++) {
      #pragma omp task depend(in: A[k][k]) depend(inout: A[i][k])
      cblas_dtrsm(..., A[k][k], TS, A[i][k], TS);
    }

    for (long i = k+1; i < N/TS; i++) {
      for (long j = k+1; j < i; j++) {
        #pragma omp task depend(in: A[i][k], A[j][k]) depend(inout: A[i][j])
        cblas_dgemm(..., A[i][k], TS, A[j][k], TS, 1.0, A[i][j], TS);
      }
      #pragma omp task depend(in: A[i][k]) depend(inout: A[i][i])
      cblas_dsyrk(..., A[i][k], TS, 1.0, A[i][i], TS);
    }
  }
  #pragma omp taskwait
}
```



Cholesky factorization: OpenMP tasks + CUDA

Cholesky factorization (**heterogeneous**)

```
void cholesky(size_t N, size_t TS, double (*A)[N/TS][TS][TS]) {
  for (long k = 0; k < N/TS; k++) {
    #pragma omp task depend(inout: A[k][k])
    LAPACKE_dpotrf(..., A[k][k], TS);

    for (long i = k+1; i < N/TS; i++) {
      #pragma omp task depend(in: A[k][k]) depend(inout: A[i][k])
      cblas_dtrsm(..., A[k][k], TS, A[i][k], TS);
    }

    for (long i = k+1; i < N/TS; i++) {
      for (long j = k+1; j < i; j++) {
        #pragma omp task depend(in: A[i][k], A[j][k]) depend(inout: A[i][j])
        cuda_dgemm(N, TS, k, i, j, ...);
      }
      #pragma omp task depend(in: A[i][k]) depend(inout: A[i][i])
      cblas_dsyrk(..., A[i][k], TS, 1.0, A[i][i], TS);
    }
  }
  #pragma omp taskwait
}
```



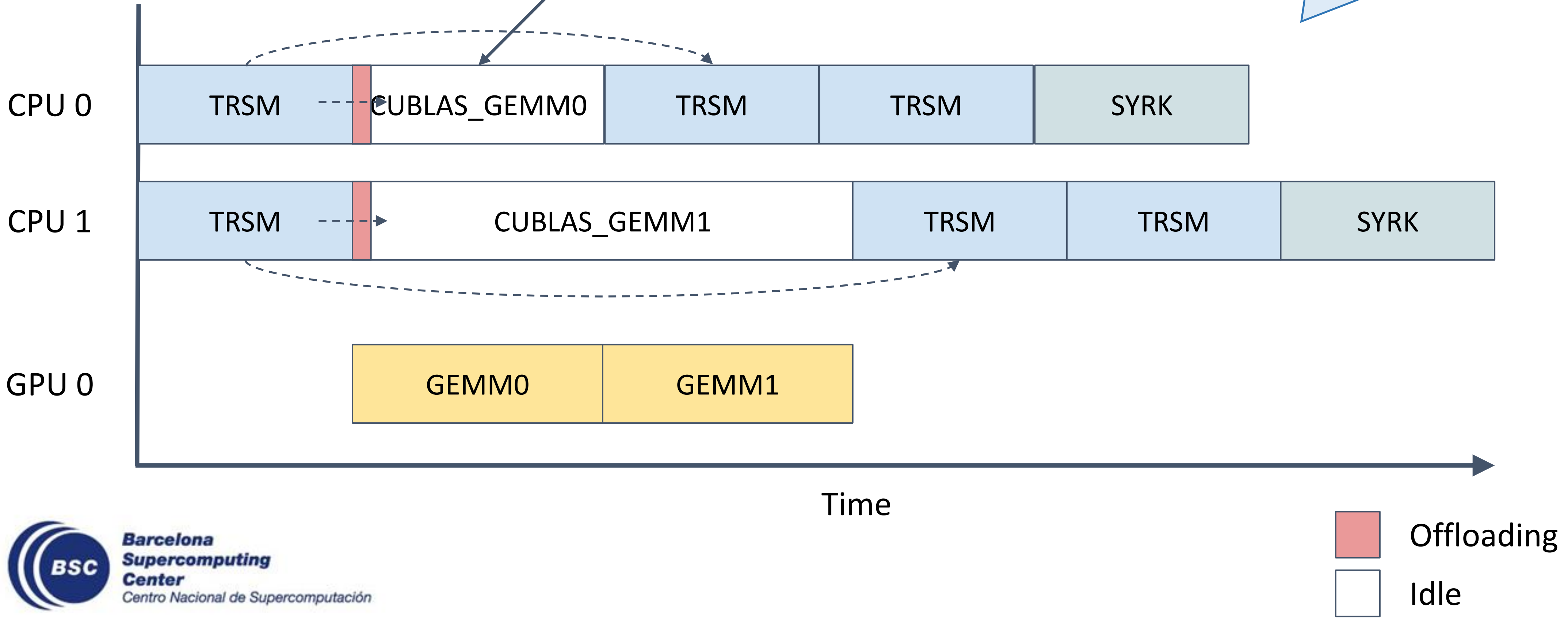
Cholesky factorization: OpenMP tasks + CUDA

```
static void cuda_dgemm(size_t N, size_t TS, long k, long i, long j,
                       double (*h_A)[N/TS][TS][TS], double (*d_A)[N/TS][TS][TS]) {
    const size_t size = TS*TS*sizeof(double);
    // Set the default context
    cuCtxSetCurrent(defaultContext);
    // Initialize cublasHandle if needed
    static __thread cublasHandle_t handle = NULL;
    if (!handle)
        cublasCreate(&handle);
    // Create a stream and bind it to the handle
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    cublasSetStream(handle, stream);
    // Asynchronously copy input to device
    cudaMemcpyAsync((void *)d_A[k][i], (const void *)h_A[k][i], size, cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync((void *)d_A[k][j], (const void *)h_A[k][j], size, cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync((void *)d_A[j][i], (const void *)h_A[j][i], size, cudaMemcpyHostToDevice, stream);
    // Launch Dgemm kernel
    const double alfa = -1.0f, beta = 1.0f;
    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, TS, TS, TS, &alfa, (const double *)d_A[k][i], TS,
                       (const double *)d_A[k][j], TS, &beta, (double *)d_A[j][i], TS);
    // Asynchronously copy output to host
    cudaMemcpyAsync((void *)h_A[j][i], (const void *)d_A[j][i], size, cudaMemcpyDeviceToHost, stream);
    // Synchronize with the stream
    cudaStreamSynchronize(stream);
    // Destroy stream and handle
    cudaStreamDestroy(stream);
}
```

Cholesky factorization: OpenMP tasks + CUDA

CPUs stuck in *cudaStreamSynchronize()* even though there is plenty of work!

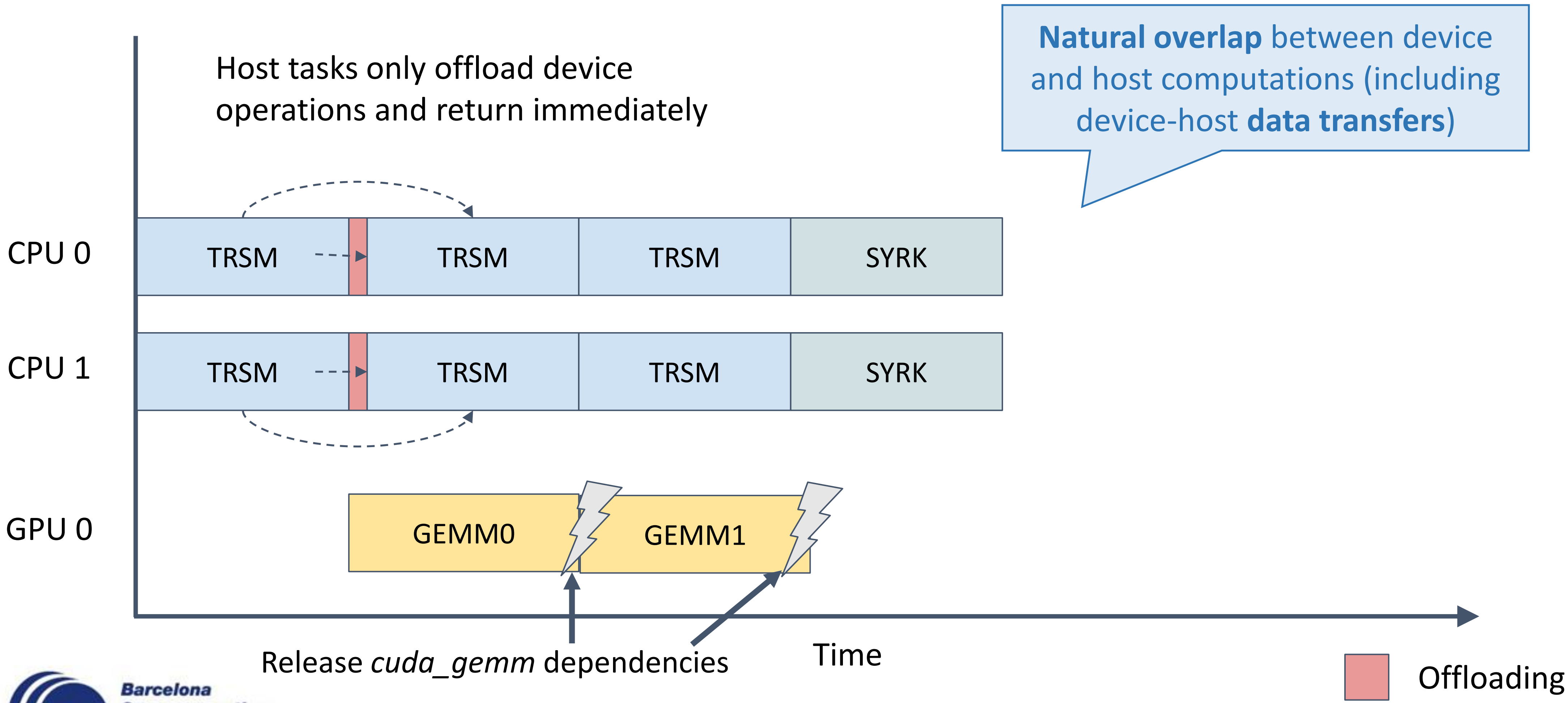
No overlap of device and host computation!



Cholesky factorization: OpenMP tasks + TACUDA

```
static void cuda_dgemm(size_t N, size_t TS, long k, long i, long j,
                       double (*h_A)[N/TS][TS][TS], double (*d_A)[N/TS][TS][TS]) {
    const size_t size = TS*TS*sizeof(double);
    //Default context set by TACUDA
    //cuCtxSetCurrent(defaultContext);
    // Initialize cublasHandle if needed
    static __thread cublasHandle_t handle = NULL;
    if (!handle)
        cublasCreate(&handle);
    // Get a stream and bind it to the handle
    cudaStream_t stream;
    tacudaGetStream(&stream);
    cublasSetStream(handle, stream);
    // Asynchronously copy input to device
    cudaMemcpyAsync((void *)d_A[k][i], (const void *)h_A[k][i], size, cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync((void *)d_A[k][j], (const void *)h_A[k][j], size, cudaMemcpyHostToDevice, stream);
    cudaMemcpyAsync((void *)d_A[j][i], (const void *)h_A[j][i], size, cudaMemcpyHostToDevice, stream);
    // Launch Dgemm kernel
    const double alfa = -1.0f, beta = 1.0f;
    cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T, TS, TS, TS, &alfa, (const double *)d_A[k][i], TS,
                       (const double *)d_A[k][j], TS, &beta, (double *)d_A[j][i], TS);
    // Asynchronously copy output to host
    cudaMemcpyAsync((void *)h_A[j][i], (const void *)d_A[j][i], size, cudaMemcpyDeviceToHost, stream);
    // Bind the task completion to stream synchronization
    tacudaSynchronizeStreamAsync(stream);
    // Return the stream
    tacudaReturnStream(stream);
}
```

OpenMP tasks + TACUDA

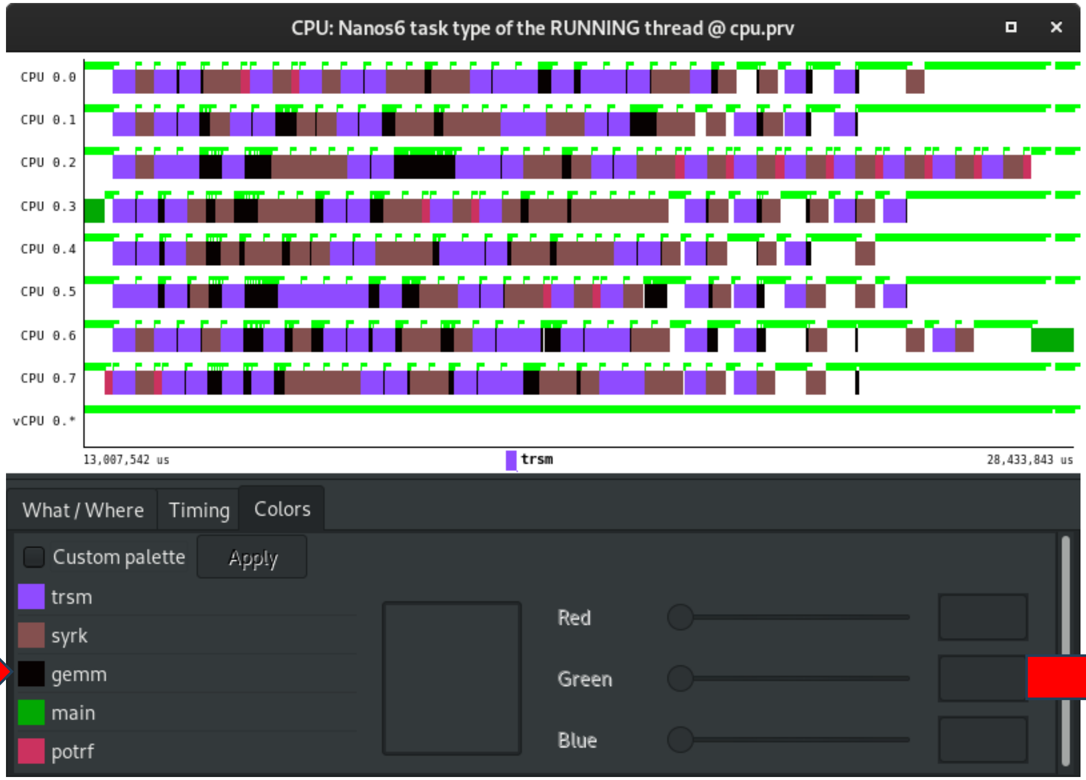


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

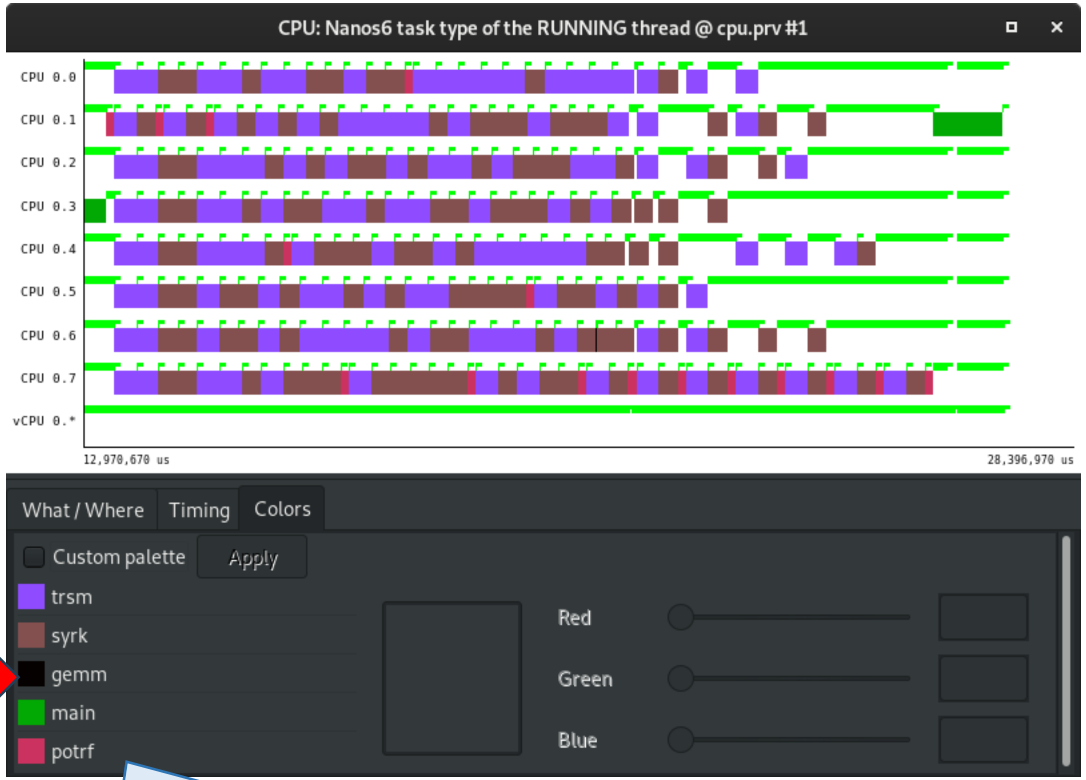
https://github.com/bsc-pm/tacuda

OpenMP tasks + TACUDA

Cholesky OpenMP + **CUDA**



Cholesky OpenMP + **TACUDA**



Gemm tasks (offloading) **consume minimal CPU** resources!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Portability of Task-Aware Libraries

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

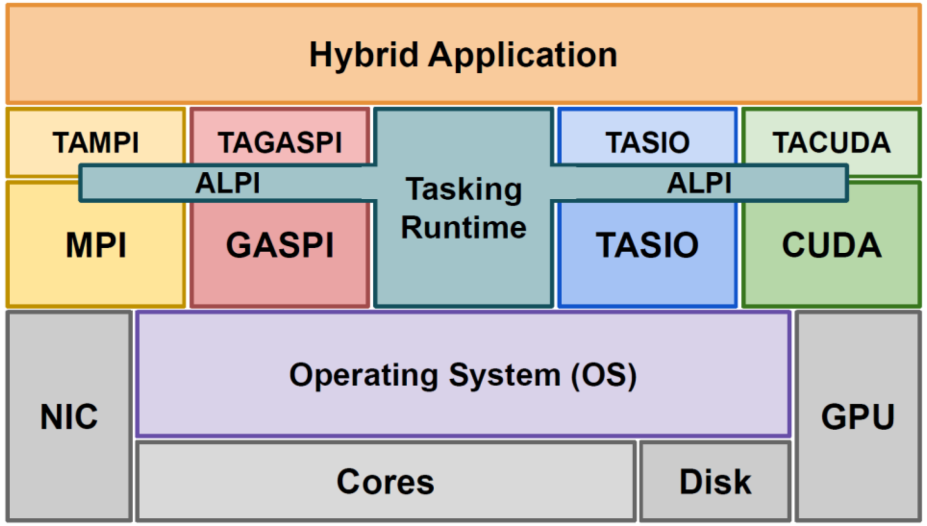
Outline

- Motivation
- Principles of Task-Awareness
- Task-Aware Libraries (TA-X)
- Task-Aware MPI (TAMPI)
- Task-Aware CUDA (TACUDA)
- **Portability and Interoperability of TA-X Libraries**

Interoperability between TA-X Libraries

- Combining **blocking** and **non-blocking** APIs from different TA-X libraries
- TA-X calls are allowed on **different tasks** or even inside the **same task**!

```
#pragma omp task depend(in: x[0;N]) depend(inout: y[0;N], d_y[0;N], d_x[0;N])
{
  MPI_Recv(&y, N, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, ...);
  MPI_Send(&x, N, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD);
  cudaMemcpyAsync(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice, stream);
  cuda_gemm_kernel<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
  cudaMemcpyAsync(y, d_y, N*sizeof(double), cudaMemcpyDeviceToHost, stream);
  tacudaStreamSynchronizeAsync(stream);
}
```

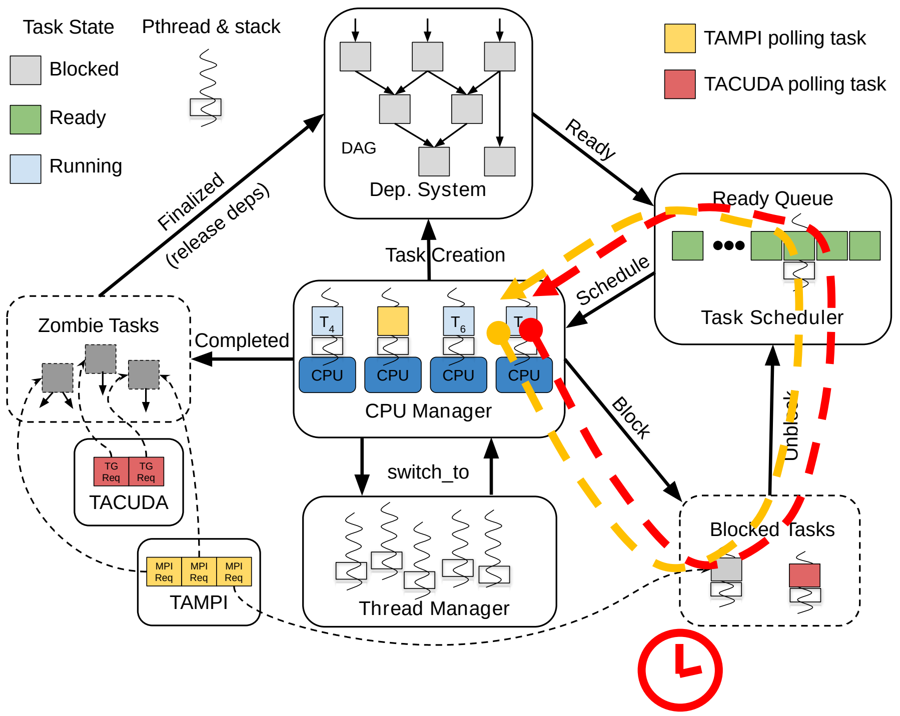


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Interoperability between TA-X Libraries

- Combining **blocking** and **non-blocking** APIs from different TA-X libraries
- TA-X calls are allowed on **different tasks** or even inside the **same task**!
  - Blocking APIs compose easily (pause/resume cycles)

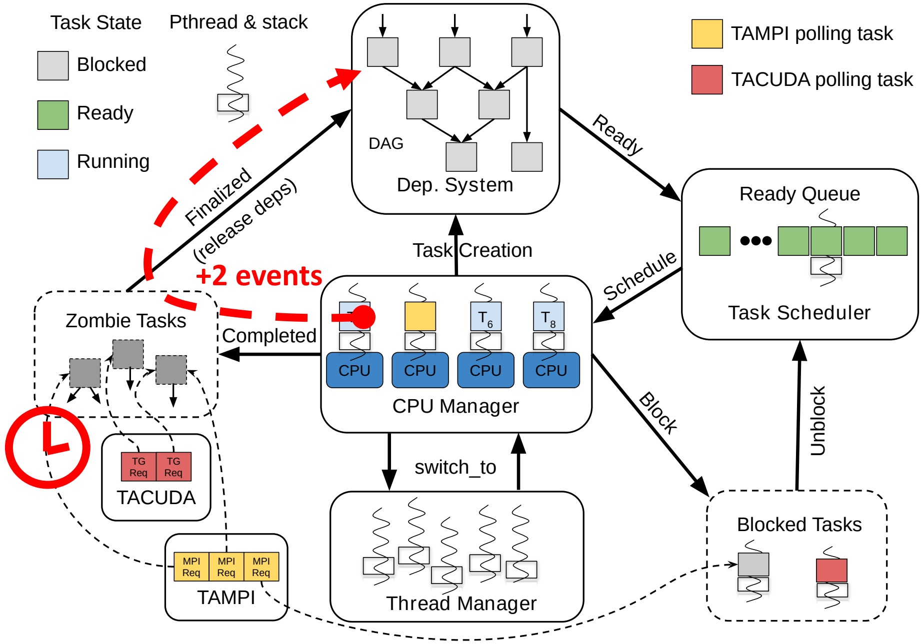
```
#pragma omp task depend(in: x[0;N]) depend(inout: y[0;N], d_y[0;N], d_x[0;N])
{
  MPI_Recv(&y, N, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, ...);
  MPI_Send(&x, N, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD);
  cudaMemcpyAsync(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_y, y, N*sizeof(double), cudaMemcpyHostToDevice, stream);
  cuda_gemm_kernel<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
  cudaMemcpyAsync(y, d_y, N*sizeof(double), cudaMemcpyDeviceToHost, stream);
  tacudaStreamSynchronizeAsync(stream);
}
```



Interoperability between TA-X Libraries

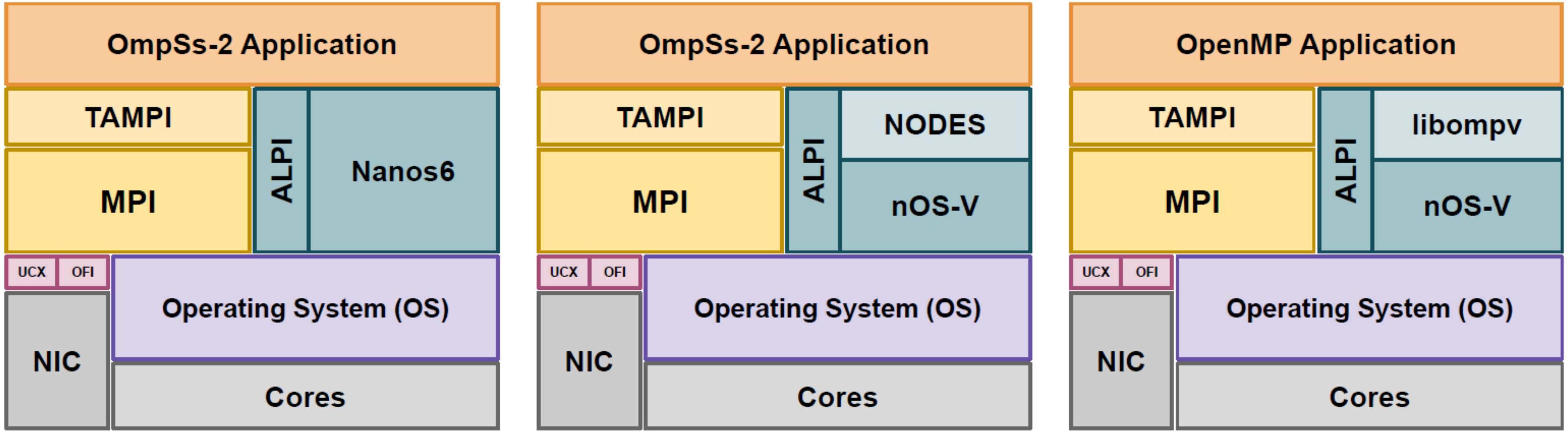
- Combining **blocking** and **non-blocking** APIs from different TA-X libraries
- TA-X calls are allowed on **different tasks** or even inside the **same task**!
  - Non-blocking APIs can also be easily combined!

```
#pragma omp task depend(in: x[0;N]) depend(inout: y[0;N], d_y[0;N], d_x[0;N])
{
  MPI_Recv(&y, N, MPI_DOUBLE, src, tag, MPI_COMM_WORLD, ...);
  TAMPI_Isend(&x, N, MPI_DOUBLE, dst, tag, MPI_COMM_WORLD);
  cudaMemcpyAsync(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_y, y, N* sizeof(double), cudaMemcpyHostToDevice, stream);
  cuda_gemm_kernel<<<numBlocks, blockSize, stream>>>(N, d_x, d_y);
  cudaMemcpyAsync(y, d_y, N* sizeof(double), cudaMemcpyDeviceToHost, stream);
  tacudaStreamSynchronizeAsync(stream);
}
```



Portability of TA-X Libraries

- Any **task-based runtime** implementing **ALPI** is compatible
- Supported models
  - **OmpSs-2** through **Nanos6** and **nOS-V** runtimes
  - **OpenMP** through the **LLVM/libompv** and **nOS-V** runtimes



(A) OmpSs-2 over Nanos6. (B) OmpSs-2 over NODES. (C) OpenMP over nOS-V.

nOS-V: https://github.com/bsc-pm/nos-v
Nanos6: https://github.com/bsc-pm/nanos6

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

Kevin Sala, Xavier Teruel and Vicenç Beltran

kevin.sala@bsc.es