



**Barcelona
Supercomputing
Center**
Centro Nacional de Supercomputación



EXCELENCIA
SEVERO
OCHOA

Extending the Task-Aware MPI (TAMPI) Library to Support Asynchronous MPI primitives

**Kevin Sala, X. Teruel, J. M. Perez,
V. Beltran, J. Labarta**

24/09/2018

OpenMPCon 2018, Barcelona

Outline

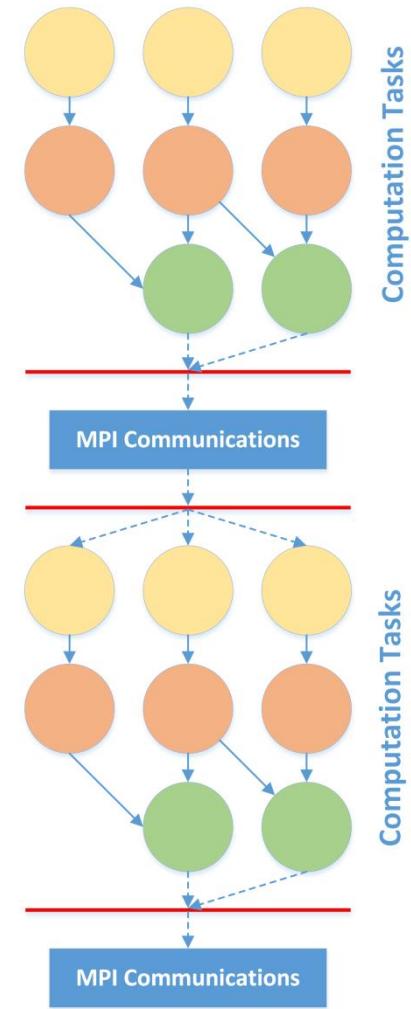
- Overview TAMPI Library
- Motivation
- Proposal for asynchronous primitives
- Task-based runtime system requirements
 - External events API
- Extending the TAMPI library
- Evaluation
 - Gauss-Seidel
- Conclusions and future work

Introduction to TAMPI

- Current HPC trends show that exascale systems will consist of
 - Huge number of **distributed** memory **nodes**
 - Each node containing a large number of **compute cores**
- Hybrid parallelization allows to exploit intra- and inter-node parallelism
 - Distributed programming models (e.g. MPI) for **inter-node** parallelism
 - Shared-mem programming models (e.g. OpenMP) for **intra-node** parallelism
- We could exploit some synergies...
 - Fine-grained synchronization across nodes
 - **Overlap** of computation and communication phases
 - Leverage intra-node parallelism to **hide** network latency and maximize network throughput
- However, these PMs were **not designed to be combined**
 - Common strategies are **suboptimal** (e.g. sequential communication)
 - Advanced techniques (e.g. double-buffering) are **difficult** to program, still **suboptimal**, and sometimes even **unfeasible**

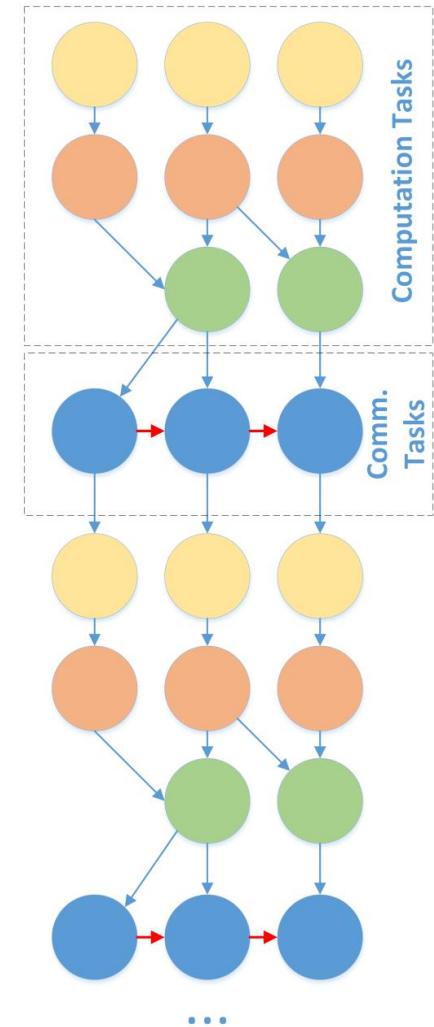
Common Fork-Join Model

- The common approach is to interleave computation and communication phases
 - Computation phases are executed in **parallel**
 - e.g. “*parallel for*” or “*tasks + taskwait*”
 - Communication phases are executed **sequentially**
- Advantages
 - **Straightforward** to implement
 - No need of MPI_THREAD_MULTIPLE
- Disadvantages
 - **No overlap** of computation and communication phases
 - Communications **delayed** until all computations done



Taskifying Communications

- In this approach *tasks* are used **both** for computational phases and communication phases
- Advantages
 - Potential to **overlap** computation and communication phases
 - **No artificial barrier** between phases
- Disadvantages
 - Communication *tasks* have to be serialized (**red arrows**)
 - Otherwise a **deadlock** could be produced !
 - These limitations severely limits the potential benefits of this approach



TAMPI

- Proposed a new MPI threading level **MPI_TASK_MULTIPLE** to enable an interoperability mechanism between MPI and the task-based programming model
 - Enabled from the user application by calling to *MPI_Init_thread*
- In applications where **tasks** perform **blocking MPI operations** (e.g. `MPI_Recv`), this mechanism prevents the underlying hardware threads that execute those tasks from **blocking** inside these functions
 - Allowing **other tasks** to be **executed**
- Also, this mechanism should:
 - Improve the **programmability** and **performance** of hybrid applications
 - **Reorder** the execution of communication tasks without a deadlock
 - Be **compatible** to any task-based programming model

TAMPI

- Safe way to initialize and use the mechanism:

```
int *serial; // Sentinel used to serialize communication tasks

int main(int argc, char **argv) {
    int provided, rank;
    MPI_Init_thread(&argc, &argv, MPI_TASK_MULTIPLE, &provided);
    if (provided == MPI_TASK_MULTIPLE) serial = (int *) 0; // Disable the sentinel
    else serial = (int *) 1; // Enable the sentinel to serialize comm. tasks

    int data[N];
    init_data(data, N);

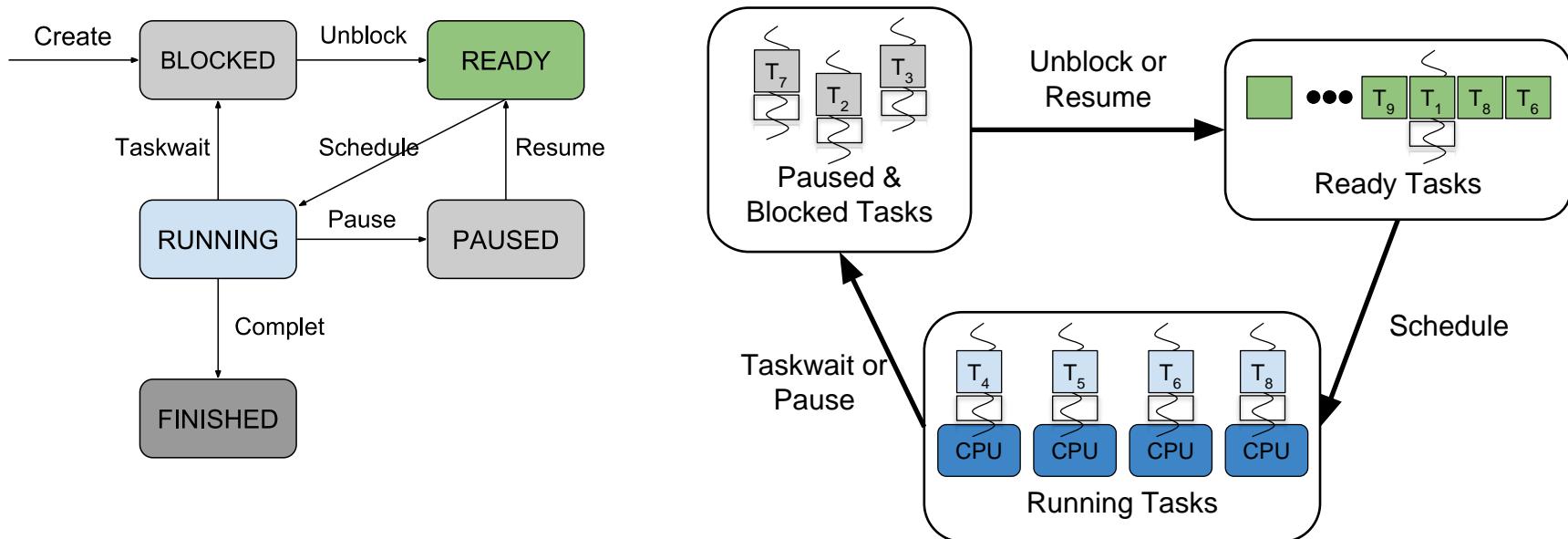
    for (int i = 0; i < N; ++i) {
        #pragma oss task in(data[i]) inout(*serial)
        communication_task(&data[i]);
    }
    #pragma oss taskwait
    MPI_Finalize();
}
```

Note: OmpSs feature. Declares a dependency on the memory region pointed by the pointer. If the pointer is NULL, the dependency is ignored.

Task Pause/Resume API

- Low-level API to programmatically pause and resume the execution of a task

```
void * get_current_blocking_context();      // Get task id  
void block_current_task(void *context);    // Block task  
void unblock_task(void *context);           // Unblock task
```



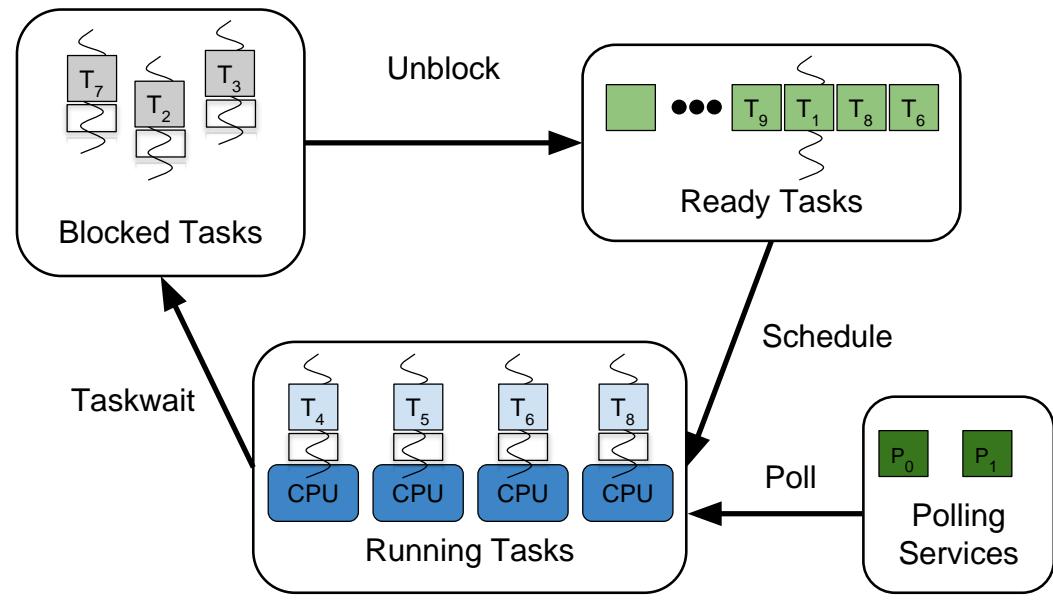
Polling Service API

- Low-level API to periodically execute an arbitrary *service*

```
// Function that the runtime calls periodically
typedef int (*polling_function_t)(void *service_data);

// Registers a function and a parameter to be called periodically
void register_polling_service(const char *service_name,
                               polling_function_t service_function, void *service_data);

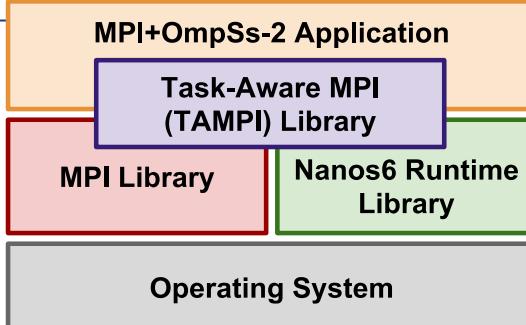
// Unregisters a function and parameter combination
void unregister_polling_service(const char *service_name,
                                 polling_function_t service_function, void *service_data);
```



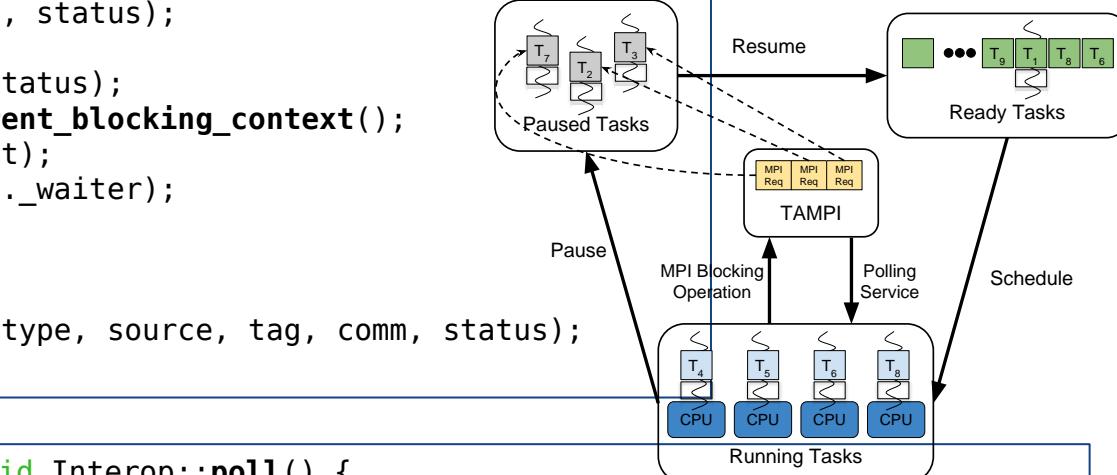
TAMPI Implementation

- Leverage the low-level pause/resume and polling service API

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status) {
    int err, completed = 0;
    if (Interop::isEnabled()) {
        MPI_Request request;
        err = MPI_Irecv(buf, count, datatype, source, tag, comm, &request);
        MPI_Test(&request, &completed, status);
        if (!completed) {
            Ticket ticket(&request, status);
            ticket._waiter = get_current_blocking_context();
            _pendingTickets.add(ticket);
            block_current_task(ticket._waiter);
        }
    }
    return err;
}
return PMMPI_Recv(buf, count, datatype, source, tag, comm, status);
}
```



```
void Interop::poll() {
    for (Ticket &ticket : _pendingTickets) {
        int completed = 0;
        MPI_Test(ticket._request, &completed, ticket._status);
        if (completed) {
            _pendingTickets.remove(ticket);
            unblock_task(ticket._waiter);
        }
    }
}
```



Outline

- Overview TAMPI Library
- **Motivation**
- Proposal for asynchronous primitives
- Task-based runtime system requirements
 - External events API
- Extending the TAMPI library
- Evaluation
 - Gauss-Seidel
- Conclusions and future work

Motivation

- The TAMPI library intercepts calls to blocking MPI primitives and **blocks** the calling tasks
 - Producing a **context-switch** !
 - Keeping alive the **stack** of the calling tasks/threads !
 - e.g. 2MB of stack per thread by default
- When **unblocking** a task, it probably won't be executed immediately
 - It could just be added into the **ready queue**
 - If the task just performs an MPI operation, it could be useless to resume the task once the operation completes
 - *Paused → Ready → Running → Finished*
 - The **release** of dependencies is **delayed** !

```
#pragma oss task out(data[i])
MPI_Recv(&data[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Motivation

- Another example...

```
#pragma oss task out(buf[0;2]) in(buf[2;2]) ...
{
    MPI_Request reqs[4];

    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

    MPI_Isend(&buf[2], 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&buf[3], 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

    // Do some work
    MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);
}
```

The task is paused and resumed, so the release of dependencies is delayed !

- All these *problems* are noticeable in applications
 - With **many communication tasks**
 - Sending/receiving **small chunks of data**

Outline

- Overview TAMPI Library
- Motivation
- **Proposal for asynchronous primitives**
- Task-based runtime system requirements
 - External events API
- Extending the TAMPI library
- Evaluation
 - Gauss-Seidel
- Conclusions and future work

Proposal for asynchronous primitives

- **TAMPI_Iwait/all**

- Asynchronous function that **links** the release of task's dependencies with the completion of the MPI requests passed as parameters
- Once the tasks **finishes** AND all MPI operations registered with **TAMPI_Iwait/all()** are **completed**, the dependencies are **released**
- Both synchronous (original) and asynchronous (extended) modes can **coexist** in the same application
 - Enabled also with **MPI_TASK_MULTIPLE**, otherwise it calls the original MPI_Wait

```
int TAMPI_Iwait(MPI_Request *request, MPI_Status *status);
int TAMPI_Iwaitall(int count, MPI_Request *requests, MPI_Status *statuses);
```

```
#pragma oss task out(buf[0]) in(buf[1]) out(statuses[0;2]) ...
{
    MPI_Request reqs[2];
    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
    MPI_Isend(&buf[1], 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[1]);
    TAMPI_Iwaitall(2, reqs, statuses);
}

#pragma oss task in(buf[0]) in(statuses[0]) ...
{
    check_status(statuses[0]);
    printf("Received integer: %d\n", buf[0]);
}
```

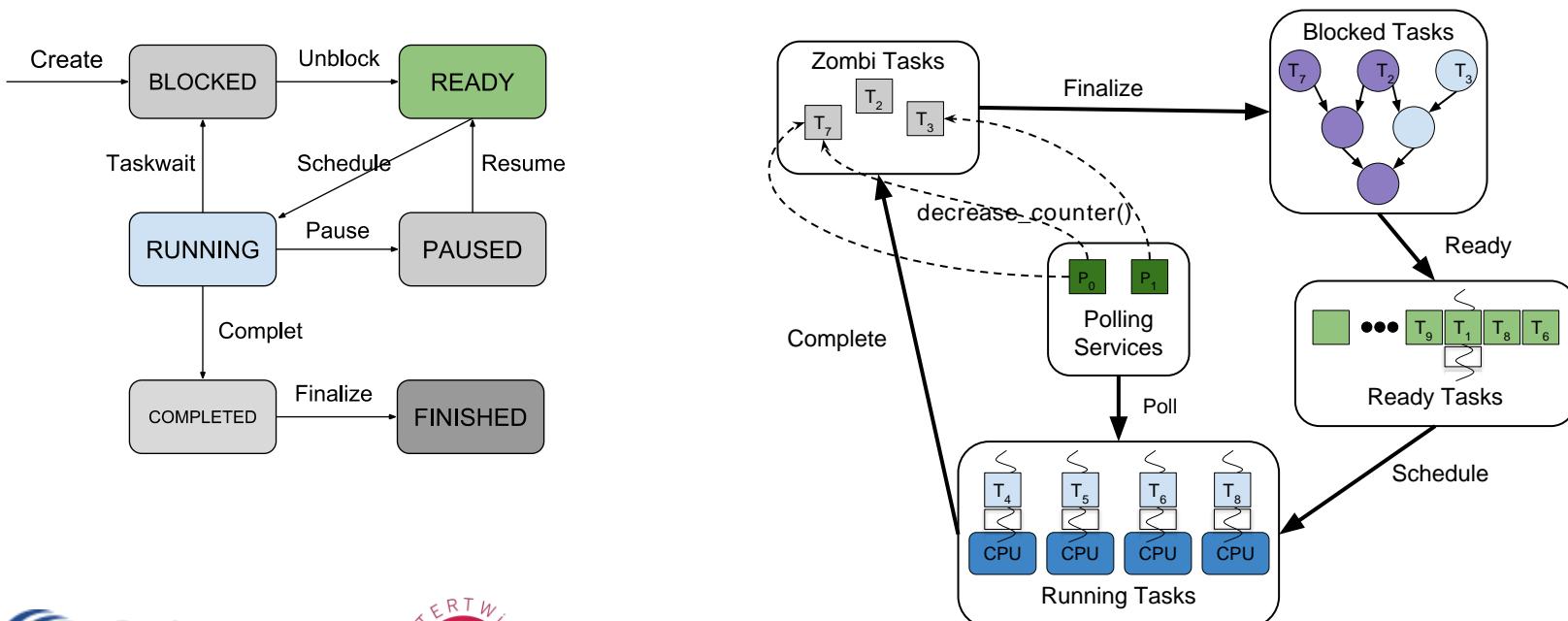
Outline

- Overview TAMPI Library
- Motivation
- Proposal for asynchronous primitives
- **Task-based runtime system requirements**
 - External events API
- Extending the TAMPI library
- Evaluation
 - Gauss-Seidel
- Conclusions and future work

External events API

- Low-level API to programmatically decouple the task completion from the release of dependencies

```
void * get_current_event_counter();
void increase_current_task_event_counter(void *event_counter, unsigned int increment);
void decrease_task_event_counter(void *event_counter , unsigned int decrement);
```



External events API

- Decouple task completion from the release of dependencies
 - All tasks have a new atomic counter initialized to 1
 - Once a task completes, the counter is decremented, and if, and only if, the counter is equal to zero the dependencies of the task are released
 - Two additional functions to add or subtract to this counter. If after decreasing the counter, it becomes 0, the dependencies are released
 - So, the dependencies of a task are released when there is no pending events and the task has finished its execution

```
class task_t {  
    /* ... */  
    std::atomic<int> events{1};  
}  
  
void task_finish(){  
    if ((current_task->events--) == 1){  
        current_task->release_dependencies();  
    }  
}
```

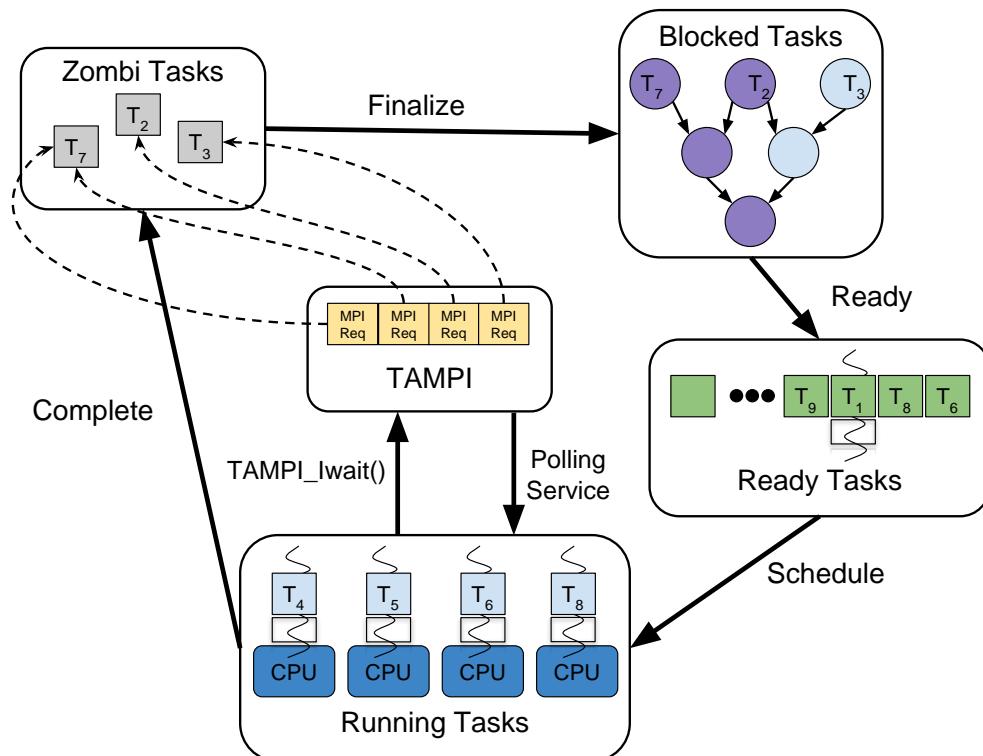
```
void * get_current_event_counter(){  
    return current_task;  
}  
  
void nanos_counter_increase(void *context, int value){  
    task_t * task = context;  
    task->events += value;  
}  
  
void nanos_counter_decrease(void *context, int value){  
    task_t * task = context;  
    if ((task->events -= value) == value){  
        task->release_dependencies();  
    }  
}
```

Outline

- Overview TAMPI Library
- Motivation
- Proposal for asynchronous primitives
- Task-based runtime system requirements
 - External events API
- **Extending the TAMPI library**
- Evaluation
 - Gauss-Seidel
- Conclusions and future work

Extending the TAMPI library

- **TAMPI_Iwait/all(...)**
 - **Asynchronous** functions that leverage the **external events** and **polling services API**
 - The event counter of the calling task is **increased** for each non-null request passed as parameter
 - Each time a request completes, the event counter of the related task is **decreased**
 - The task-based runtime system should **automatically release** the dependencies once the counter becomes zero



Extending the TAMPI library

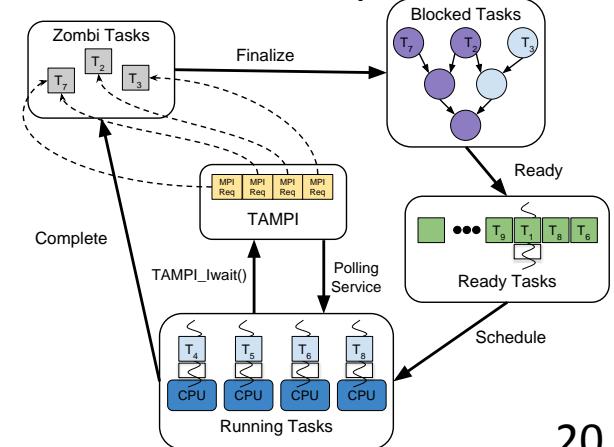
```
int MPI_Recv(...) {
    int err, completed = 0;
    if (Interop::isEnabled()) {
        MPI_Request request;
        err = MPI_Irecv(buf, count, datatype, source, tag, comm, &request);
        MPI_Test(&request, &completed, status);
        if (!completed) {
            Ticket ticket(&request, status);
            ticket._waiter = get_current_blocking_context();
            _pendingTickets.add(ticket);
            block_current_task(ticket._waiter);
        }
        return err;
    }
    return PMPI_Recv(buf, count, datatype, source, tag, comm, status);
}
```

Original TAMPI

```
int TAMPI_Iwait(MPI_Request *req, MPI_Status *status) {
    if (Interop::isEnabled()) {
        int completed = 0;
        int err = MPI_Test(req, &completed, status);
        if (!completed) {
            Ticket ticket = new Ticket(&request, status);
            ticket._cnt = get_current_task_event_counter();
            increase_current_task_event_counter(ticket._cnt, 1);
            _pendingTicket.add(ticket);
        }
        return err;
    }
    return PMPI_Wait(req, status);
}
```

Extended TAMPI

Cannot be stored at the **stack**. This function **returns immediately!**



Extending the TAMPI library

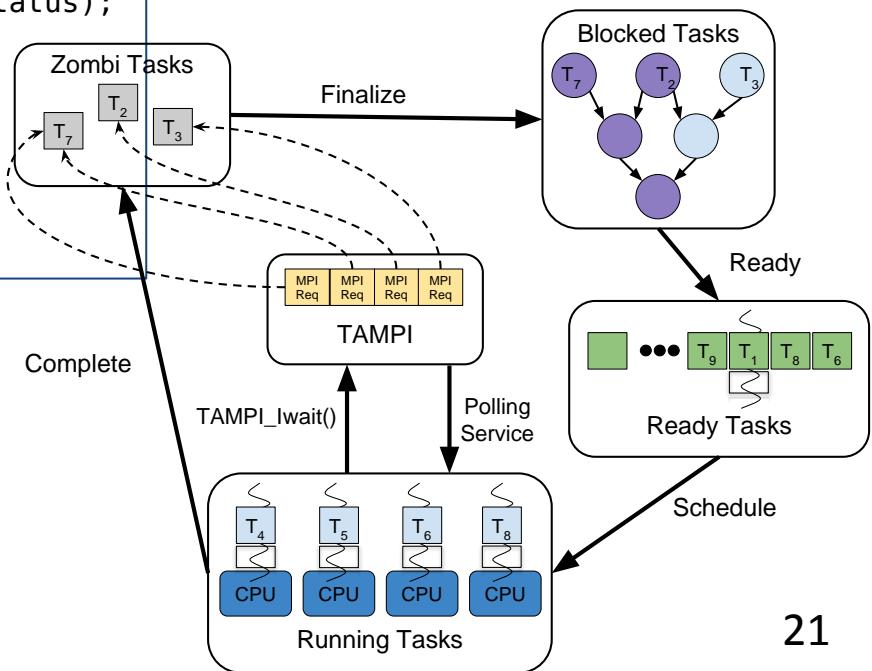
```
void Interop::poll() {
    for (Ticket &ticket : _pendingTickets) {
        int completed = 0;
        MPI_Test(ticket._request, &completed, ticket._status);
        if (completed) {
            _pendingTickets.remove(ticket);
            unblock_task(ticket._waiter);
        }
    }
}
```

Original TAMPI

```
void Interop::poll() {
    for (Ticket &ticket : _pendingTickets) {
        int completed = 0;
        MPI_Test(ticket._request, &completed, ticket._status);
        if (completed) {
            _pendingTickets.remove(ticket);
            decrease_task_event_counter(ticket._cnt, 1);
            delete ticket;
        }
    }
}
```

Extended TAMPI

Must be *freed* here!



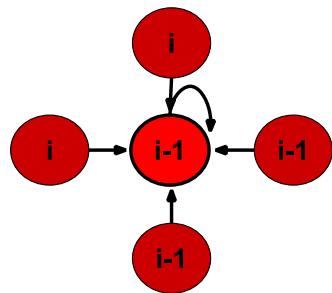
Outline

- Overview TAMPI Library
- Motivation
- Proposal for asynchronous primitives
- Task-based runtime system requirements
 - External events API
- Extending the TAMPI library
- **Evaluation**
 - Gauss-Seidel
- Conclusions and future work

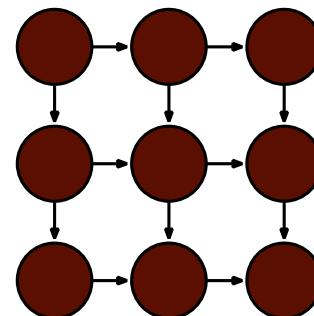
Tiled Gauss-Seidel

- In-place iterative algorithm to solve the Heat equation in a 2-D matrix
- Ex: 3 x 3 tile domain
- Each tile depend on top and left tile from current iteration and right and bottom tile from previous iteration

0	1	2
3	4	5
6	7	8



Task that computes a block
on the i -th iteration

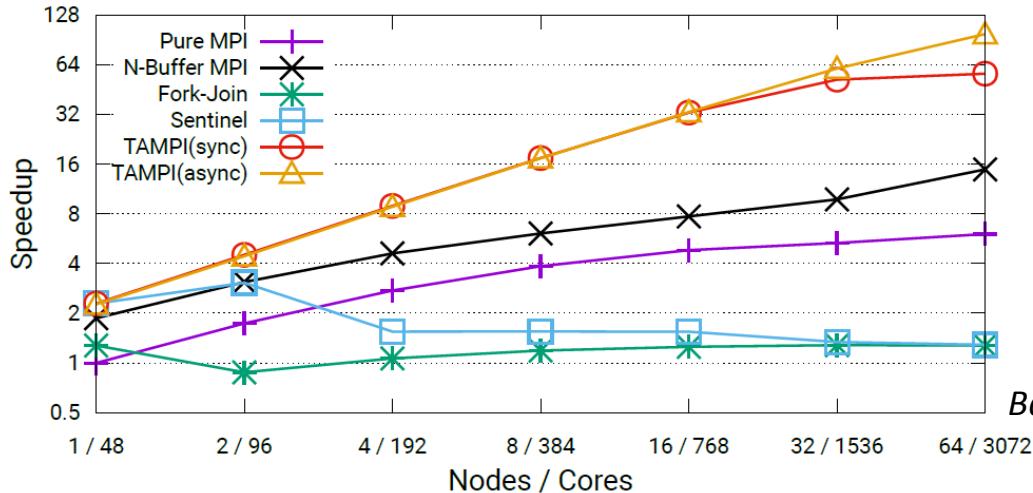


Tiled Gauss-Seidel

- **Pure MPI:** Simple MPI version. Each rank gets a consecutive set of rows. Boundary exchanges occur when the computation phase has finished
- **N-Buffer MPI:** Complex MPI version. Each rank holds several horizontal blocks. Block boundary exchanges occur **as soon as possible**
- **Fork-Join:** Simple hybrid MPI + OmpSs-2 version. Parallel computation and **sequential communication**
- **Sentinel:** Hybrid MPI + OmpSs-2. A **task** is created to **compute** each block, and another is created to **send/recv** each block boundary. **Communication tasks are serialized** by an artificial dependency
- **Interop(sync):** Same as *Sentinel* but **removing** the artificial **dependencies** of communication tasks. Enables the original interoperability mechanism
- **Interop(async):** Similar to *Sentinel* but **removing** the artificial **dependencies** of communication tasks. Uses **non-blocking** send/recv and **TAMPI_lwait**.

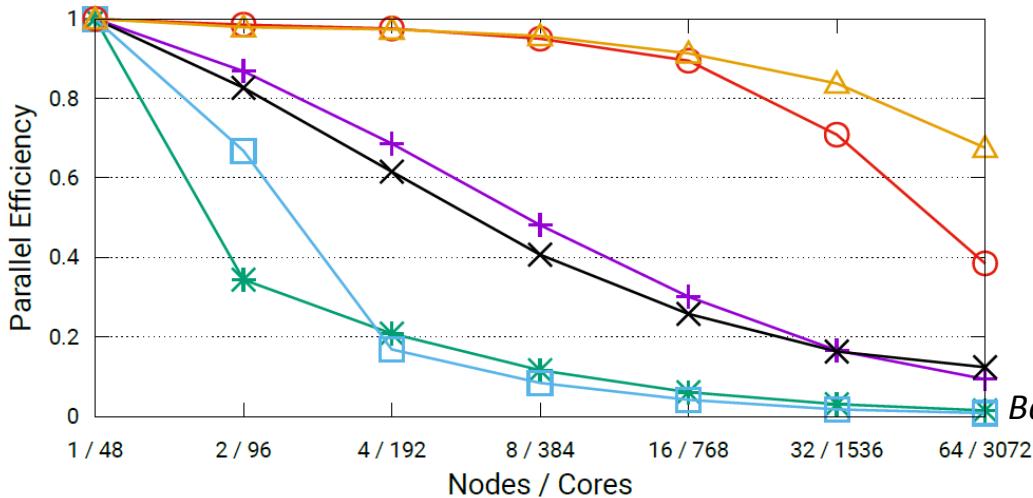


Tiled Gauss-Seidel: Strong Scaling



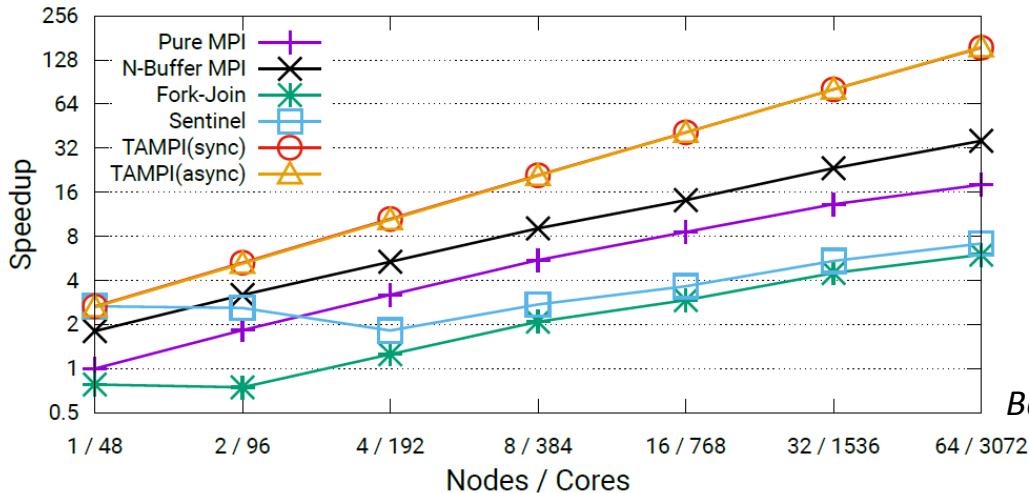
- Run on *Marenostrum4* (48 cores/node)
- $64K^2$ matrix, 1000its
- Pure/N-Buffer MPI: 48 ranks/node
- Hybrids: 1 rank/node, 48 cores/rank
- All BS=1024, Interop(async) BS=512

Baseline: Pure MPI



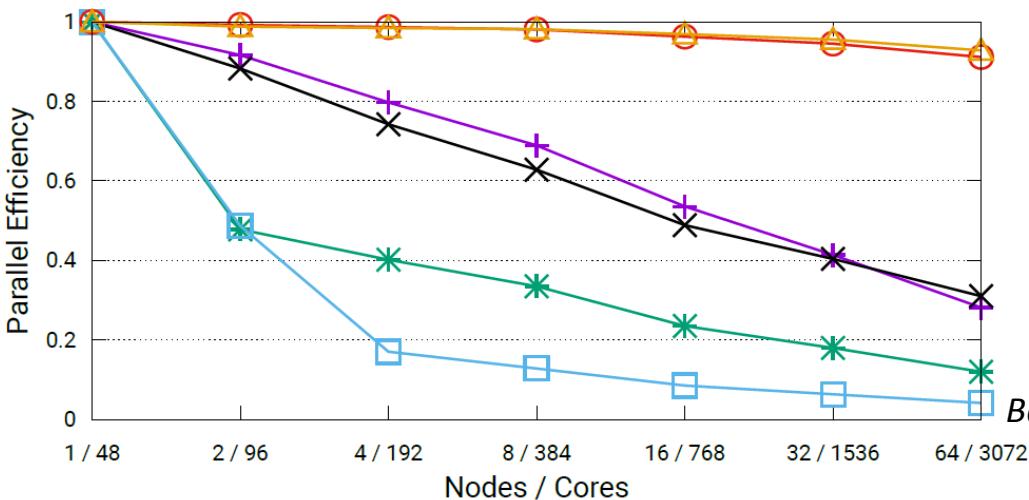
Baseline: Its own with 1 node

Tiled Gauss-Seidel: Weak Scaling



- Run on *Marenostrum4* (48 cores/node)
- $32K^2$ per node, 1000its
- Pure/N-Buffer MPI: 48 ranks/node
- Hybrids: 1 rank/node, 48 cores/rank
- All BS=1024, Interop(async) BS=512

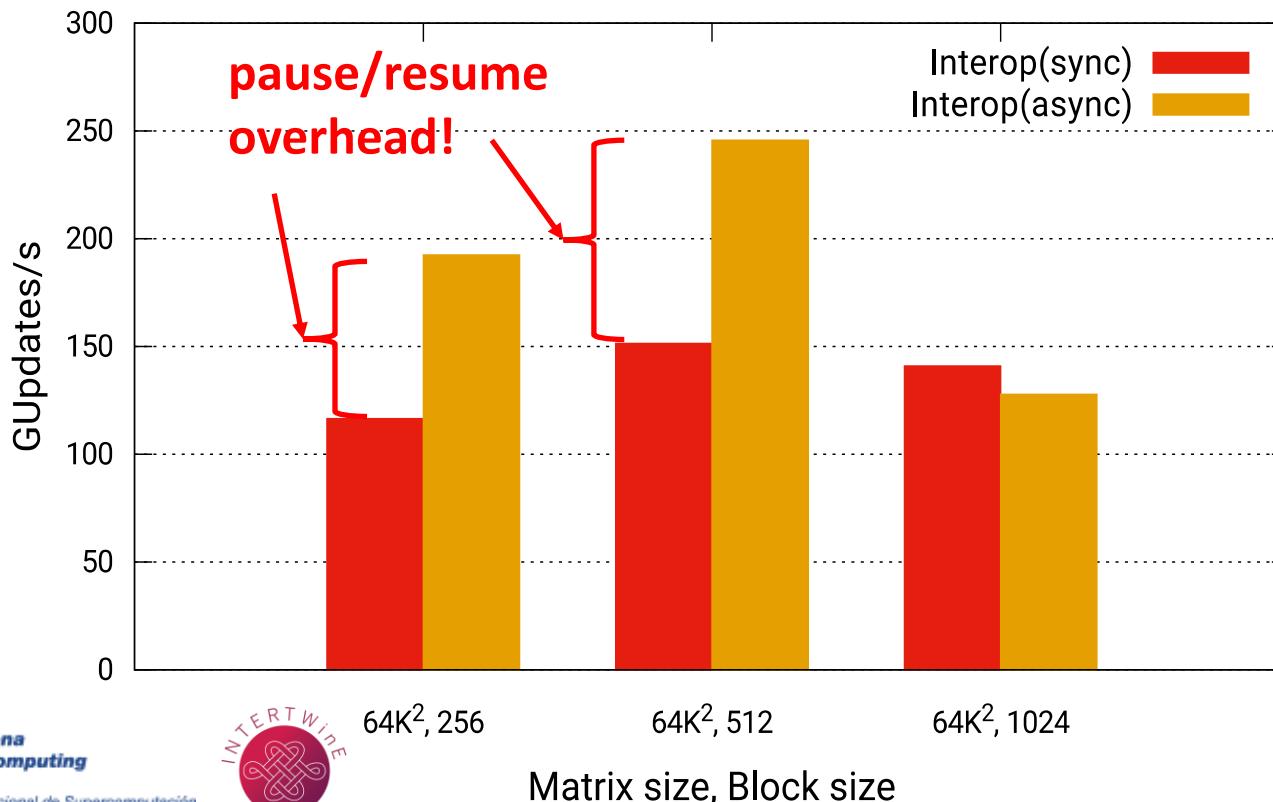
Baseline: Pure MPI



Baseline: Its own with 1 node

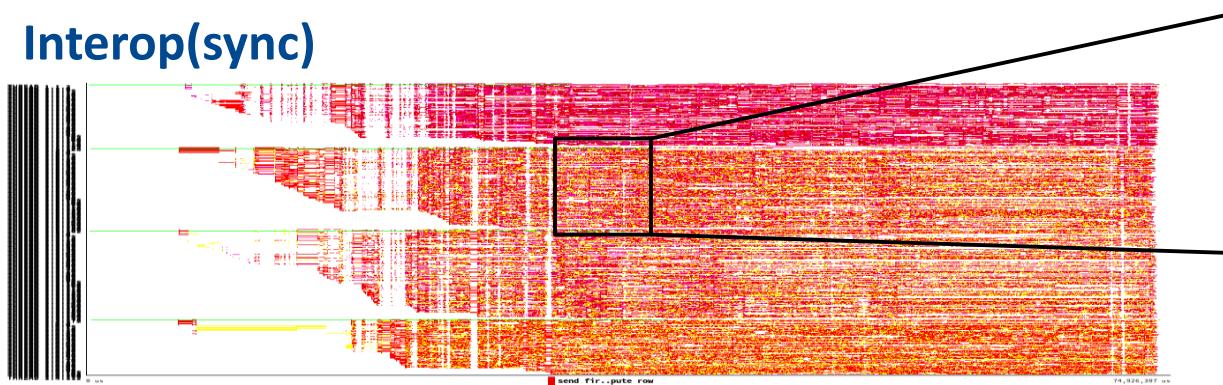
Interop(sync) vs Interop(async)

- Performance evaluation using Gauss-Seidel method
 - Problem size: 64Kx64K elements, 1500 iterations, **64 nodes** (3072 cores)
 - Block Size (BS): 256x256, 512x512 and 1024x1024
 - **MPI_Send/MPI_Recv** vs **MPI_Isend/MPI_Irecv + TAMPI_Iwait()**



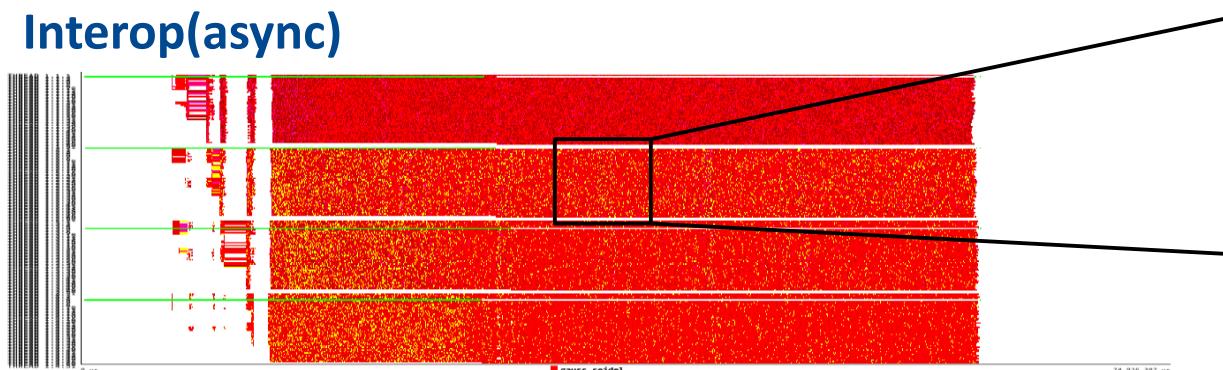
Interop(sync) vs Interop(async)

Interop(sync)



126 threads / proc

Interop(async)



51 threads / proc

Each row represents the timeline of a **created** thread

- 32Kx32K matrix, 1000 timesteps
- 4 MPI procs, 48 cores/proc
- BS=1024x1024

Outline

- Overview TAMPI Library
- Motivation
- Proposal for asynchronous primitives
- Task-based runtime system requirements
 - External events API
- Extending the TAMPI library
- Evaluation
 - Gauss-Seidel
- **Conclusions and future work**

Conclusions

- Benefits of the proposed mechanism:
 - Reduce the implicit *overhead* present in the original interoperability mechanism
 - Applications with **many communication** tasks
 - Applications sending/receiving **small** chunks of data
 - Avoid unnecessary **context switches**
 - Reduce the number of alive thread **stacks** in a given moment
 - Both interoperability mechanisms can **coexist** in the same application

Future Work

- Continue with the evaluation of TAMPI(async)
- Add support for MPI one sided API
- Task-Aware GASPI (TAGASPI) library (WiP)



**Barcelona
Supercomputing
Center**
Centro Nacional de Supercomputación



EXCELENCIA
SEVERO
OCHOA

THANK YOU!

ksala@bsc.es

www.bsc.es