UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

TASK-AWARE LPF: INTEGRATING A MODEL-COMPLIANT COMMUNICATION LAYER WITH TASK-BASED PROGRAMMING MODELS

ARNAU CINCA ROCA

Thesis supervisors: VICENÇ BELTRAN (BSC-CNS)
KEVIN SALA (BSC-CNS)

Tutor: DAVID ÁLVAREZ ROBERT (Department of Computer Architecture)

Degree: Master Degree in Innovation and Research in Informatics (High Performance Computing)

Thesis report

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

26/06/2023

Acknowledgments

I am immensely grateful to my advisors, Vicenç Beltran and Kevin Sala, for their unwavering support and invaluable guidance, without which this project would not have been possible.

Special thanks to my tutor, David Álvarez, for his assistance throughout.

I would also like to thank all my colleagues at BSC, particularly the System Tools and Advanced Runtimes (STAR) group, for having a fantastic work environment that has contributed to my growth and overall experience.

Last but not least, I want to express my gratitude to my family and friends for their unconditional support and encouragement throughout this journey. Their belief in me has been a constant source of motivation.

Abstract

The rapid advancement of high-performance computing (HPC) systems has led to the emergence of exascale computing, characterized by distributed memory nodes and high parallel computing capabilities. To effectively utilize these systems, the HPC community has embraced programming models that harness both inter-node and intra-node parallelism.

Inter-node parallelism is typically addressed using distributed-memory programming models like MPI and GASPI, while intra-node parallelism is exploited through shared-memory programming models such as OpenMP and OmpSs-2. However, the two-sided communication model used in MPI, which requires both the sender and receiver processes to post an operation, can impose performance limitations due to the inherent synchronization. In contrast, one-sided communication models like GASPI and Lightweight Parallel Foundations (LPF) leverage modern network fabric features and remote direct memory access (RDMA) to efficiently exchange data in distributed memory systems without the need for explicit receive operations.

In this project, we combine the Bulk Synchronous Parallel (BSP) model of LPF with the data-flow model of OmpSs-2 to exploit parallelism at both intra-node and inter-node levels. This approach maintains the simplicity of the BSP model and the performance of the data-flow model. By enabling optimal overlap between computation, communication, and synchronization phases, we effectively utilize available resources. The flexibility of the data-flow model allows for adjusting computation tasks that are not tightly bound to BSP model phases, facilitating early or delayed execution based on resource availability.

To optimize the BSP model, new zero-cost synchronization methods are designed, improving performance and flexibility. These methods offer localized synchronization but require a fixed communication pattern or user-defined criteria, limiting programmability. Additionally, bi-directional communication is often required, necessitating the inclusion of empty messages in applications without bi-directional communication.

Our implementation is evaluated against Task-Aware MPI (TAMPI), demonstrating that with a single coarse-grained synchronization primitive, we can still hide synchronization overheads and reach competitive performance. The results show that the zero-cost synchronization methods perform similarly to TAMPI, indicating that coarse synchronization is sufficient for iterative applications. The evaluation highlights the effectiveness of the proposed approach in improving performance and programmability in HPC applications.

Table of Contents

List of Figures

List of Tables

List of Code Samples

1 | Introduction

The rapid advancement of high-performance computing (HPC) systems has paved the way for the emergence of exascale computing. These systems are characterized by numerous distributed memory nodes, each housing multiple compute cores, providing high parallel computing capabilities.

In response to this trend, the HPC community has embraced new programming models that effectively harness both inter-node and intra-node parallelism [1–3]. To approach inter-node parallelism, distributed-memory programming models, like MPI [4]or GASPI [5], have been widely adopted. Concurrently, shared-memory programming models such as OpenMP [6] or OmpSs-2 [7] have proven instrumental in exploiting the potential of intra-node parallelism.

There are different approaches in the inter-node parallelism models. The most commonly used is the two-sided communication model, with MPI as the current standard. But this model may limit the performance, as both the sender and the receiver need to post their respective operations to establish communication. Recent studies expect that this model may not suffice for the exascale systems and beyond. To overcome this limitation the one-sided communication models, such as GASPI or Lightweight Parallel Foundations (LPF) [8], exploits the modern network fabric features to efficiently exchange data in distributed memory systems, avoiding the need for the receiver to post a receive operation by using remote direct memory access (RDMA).

To exploit the intra-node parallelism, the current standard is OpenMP, which includes several parallel programming models. The communication in these models is usually based on the exploit of the shared memory that the system already has. Of these programming models, the most promising is the data-flow programming model, which transparently enables overlapping the execution of tasks from different application phases, allowing the maximum possible usage of the resources.

There are different approaches to combine intra- and inter-node parallelism, the most common approach is the fork-join parallelization strategy, where it starts a parallel region to execute the computation phase and closes the parallel region in the communication phase, executing the communication in serial. Although this is an easy approach, it presents some performance problems. One problem is that the parallelism may be obstructed by the synchronization enforced among computation phases and nodes, this synchronization also can introduce some overhead. Another problem of this approach is the lack of overlap between the computation and communication

phases. Finally, this approach only works when the synchronization in the communication primitives is granted, which in one-sided communication is rarely the case, enforcing the application to explicitly add synchronization primitives.

To grant the synchronization in the communication primitives, the Bulk synchronous parallel (BSP) model adds a synchronization phase, which enforces the completion of all the communication before continuing with the execution. The BSP model consists of three phases, the computation phase, the communication phase, and the synchronization phase. In the computation phase, each process performs its local computation, usually a subset of the problem. In the communication phase, each processor exchanges the new computed data. The synchronization phase acts as a barrier and ensures that every process has finished the communication phase before starting another computation phase. The BSP model does not specifically defines intra-node parallelism in the computation phase, but it is easy to combine with the fork-join strategy. However, this model solves the communication synchronization in a very coarse manner by enforcing the synchronization of all the communication in one phase, this makes the model very sensitive to load imbalances in the computation and communication phases.

Another approach is integrating the communication phase and the computation phase into a data-flow model using tasks, where the main complexity comes from defining correctly the tasks' dependencies to deal with the synchronizations between tasks. However, current implementations of the inter-node parallel models have little or no support to be executed in a highly concurrent and asynchronous way. Inter-node libraries such as MPI and GASPI provide support for concurrent invocations from multiple threads. But this support does not suffice with the current task-based programming models such as OpenMP or OmpSs-2, as these models are not aware of the synchronization of the communication primitives. There has been some research to address these interoperability issues. The Task-Aware MPI (TAMPI) [9, 10] and Task-Aware GASPI (TAGASPI) [11, 12] provide the interoperability mechanisms to allow the efficient taskification of MPI and GASPI communication operations, respectively.

The current implementation of Light Parallel Foundations (LPF), which is a BSP model, does not support task-based models, limiting the library to use at most fork-join strategies, where it cannot reach optimal performance. For this reason, this work studies how to integrate and exploit synergies between the LPF one-sided primitives and the OmpSs-2 data-flow execution model, which will be used to relax the strict BSP model and enable overlapping computation and communication phases. In this thesis, (1) we design and implement the new Task-Aware LPF (TALPF) library, which combines the OmpSs-2 data-flow model with the LPF one-sided primitives and model-compliance; and (2) we compare TALPF against TAMPI.

1.1 Objectives

The main objective of this work is to integrate a data-flow model, such as OmpSs-2, with a BSP model, to achieve a data-flow model performance, while keeping the simplicity of the BSP model.

More specifically, the objectives of this work are:

- Integrate all three BSP phases (computation, communication, and synchronization) of LPF using the data-flow model of OmpSs-2. This integration will enhance the coordination between these phases, allowing for the efficient execution of parallel tasks.
- Relax the synchronization phase to allow some tasks, that do not need to follow strictly the BSP model, to move between synchronizations. This flexibility will provide more freedom in task scheduling, potentially improving overall performance.
- Design and implement the zero-cost synchronization methods, to improve the flexibility of the synchronization between processes. These methods will minimize the overhead associated with synchronization, leading to better performance and resource utilization.

1.2 Contributions

With the new method presented in this thesis, we are able to:

- Taskify all three phases of LPF allowing multiple communication operations to execute in parallel without breaking the BSP model. Also allowing the computation tasks to be executed in parallel with the communication phase or even the synchronization phase.
- Provide a more relaxed synchronization, allowing more flexibility in the communication and computation tasks that are not strictly bound to the BSP structure (intra-node).
- Provide new synchronization methods that improve the performance by allowing more flexibility and less extra communication between processes (inter-node).

1.3 Document Structure

This thesis is structured as follows. Chapter 2 introduces all the basic components that compose TALPF. Chapter 3 gives a brief scope of previous task-aware libraries. Chapter 4 lists all the tools and methodologies that made this thesis possible. In chapter 5 we define the design choices that had been made over LPF to allow task awareness and further optimizations. Chapter 6 we explain the implementation of the TALPF library, going into all the implementation details. In chapter 7 we evaluate the performance of the TALPF implementation, and we compare it against TAMPI. Finally, in chapters 8 and 9 we conclude this thesis and give an insight into possible future work.

2 | Background

In this chapter, we will introduce the components that compose Task-Aware LPF. These components are the original Light Parallel Foundations (LPF) library, Message-Passing Interface (MPI), IBVerbs, and OmpSs-2.

Light Parallel Foundations library is used as the base structure that we will adapt to be task aware. MPI serves as a communication library for the setup and auxiliary communications. IBVervs is used as the primary communication library that uses the communication primitives. Finally, OmpSs-2 will be used to taskify the computation, communication, and synchronization phases.

2.1 Message Passing Interface (MPI)

MPI (Message Passing Interface) [4] is a widely recognized specification for a message-passing library interface. This specification has been developed collaboratively by a community of parallel computing vendors, computer scientists, and application developers, following an open process. Its primary focus is on the message-passing parallel programming model, where data is exchanged between processes through cooperative operations. Additionally, MPI offers extensions beyond this model, including collective operations, remote-memory access (RMA) operations, dynamic process creation, and parallel I/O.

It is important to note that MPI is not a programming language but rather a library interface. All MPI operations are expressed as functions, subroutines, or methods, depending on the language bindings used. The MPI standard provides support for C and Fortran bindings, ensuring compatibility across different programming languages.

One of the main advantages of establishing a message-passing standard like MPI is the enhanced portability and ease of use it brings. By building higher-level routines and abstractions on top of the lower-level message-passing routines, the benefits of standardization become particularly evident in distributed-memory communication environments. Furthermore, a message-passing standard greatly benefits vendors, as it defines a clear set of routines that can be efficiently implemented on their system platforms.

The Message-Passing Interface (MPI) is designed to achieve several key objectives, which include:

- **Designing an application programming interface (API):** MPI aims to provide a well-defined interface for developers to write parallel applications using message-passing techniques.
- **Enabling efficient communication:** MPI strives to optimize communication by minimizing memory-to-memory copying, allowing computation and communication to overlap, and leveraging communication co-processors whenever possible.
- **Supporting heterogeneous environments:** MPI is designed to work in environments where different types of processors or architectures are interconnected.
- **Providing C and Fortran bindings:** MPI ensures that developers can easily use the interface in both C and Fortran programming languages, making it accessible to a wide range of users.
- **Assuming reliable communication:** MPI abstracts the underlying communication infrastructure, handling communication failures internally and relieving the user from managing such issues.
- **Facilitating implementation on various platforms:** MPI aims to define an interface that can be effectively implemented on multiple vendor platforms, providing flexibility and interoperability.
- **Language independence:** The semantics of the MPI interface is designed to be independent of any specific programming language, allowing developers to use MPI with different language bindings while maintaining consistent behavior and functionality.
- **Support multiple thread safety modes:** The interface should be designed to accommodate various levels of thread safety, allowing concurrent execution in different threading models.

2.1.1 Initialization

The first step in any MPI application is to initialize the library. To initialize the library the user simply has to call the `MPI_Init` primitive with optionally the arguments of the application.

```
int MPI_Init(int *argc, char ***argv);
```

MPI also offers the primitive `MPI_Init_thread` to define different levels of thread safety. This call requires the sem parameters as the basic one but with the addition of the desired thread safety level, and a pointer that will be used to return the thread level that provides MPI. If MPI cannot provide the desired thread safety level, will provide the higher mode available above the desired.

The thread safety levels are:

- **MPI_THREAD_SINGLE:** Indicates that only one thread is executed.
- **MPI_THREAD_FUNNELED:** Indicates that the process may be multi-threaded, but only one thread will call MPI.
- **MPI_THREAD_SERIALIZED:** Indicates that multiple threads in the process can call MPI primitives, but not at the same time.
- **MPI_THREAD_MULTIPLE:** Indicates that multiple threads in the process can call MPI primitives with no restrictions.

```
int MPI_Init_thread(int *argc, char ***argv, int desired, int *provided);
```

Once initialized the user can retrieve information on the rank and number of ranks by using the `MPI_Comm_rank` and `MPI_Comm_size` respectively.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Comm_size(MPI_Comm comm, int *size);
```

2.1.2 Point-to-Point Communication

One of the primary functionalities of MPI is point-to-point communication, which involves both the source and destination processes. The source process is responsible for initiating the communication by invoking the sending procedure, while the destination process must call the receive procedure to receive the message correctly. It is crucial for both processes to engage in these operations, as failing to do so can lead to incomplete communication. Furthermore, point-to-point messages are uniquely identified by the sender's rank, an integer tag, and the associated communicator. This allows the receiving process to differentiate between multiple messages originating from the same sender.

In the following subsections, we will explore the primary procedures for sending and receiving data using both blocking and non-blocking modes, demonstrating the various ways in which data can be exchanged between MPI processes.

2.1.2.1 Blocking Operations

To initiate the communication the user has to call the `MPI_Send` primitive from the sender rank, with a pointer to the data, the length of that data, the type, the destination, a tag, and the communicator.

```
int MPI_Send(
       void *buf, int count,
       MPI_Datatype datatype, int dest,
       int tag, MPI_Comm comm
);
```

The user has to post a receiving operation in the destination rank. The user can use the `MPI_Recv` primitive to receive operation, with a pointer where it will receive the data, the length of the data, the source, a tag, the MPI communicator, and an optional status pointer which will be used to know the status of the operation once finished. For correct behavior of the communication, the parameters of this call have to match the parameters of the `MPI_Send`.

```
int MPI_Recv(
       void *buf, int count,
       MPI_Datatype datatype, int source,
       int tag, MPI_Comm comm, MPI_Status *status
);
```

2.1.2.2 Non-Blocking Operations

As communication can be very slow, having an operation that posts a communication operation and waits for the completion might not be always the best performant approach. With this idea in mind, MPI offers a non-blocking counterpart of almost all communication primitives. The non-blocking primitives start the communication operations but do not wait for the completion of them, to do so has extra primitives to check or wait for the completions.

The non-bocking version of the `MPI_Send`, is the `MPI_Isend` where it has the same parameters but with the addition of a `MPI_Request` which we will use to check the completion of the communication operation.

```
int MPI_Isend(
       void *buf, int count,
       MPI_Datatype datatype, int dest,
       int tag, MPI_Comm comm, MPI_Request *request
);
```

Similarly, we have the non-blocking receive operation, `MPI_Irecv`, where the parameters are the same except there is no status, and has a `MPI_Request` instead. There is no status, as at the point of ending the call the operation probably is still in the air, so there is no completion state yet.

```
int MPI_Irecv(
       void *buf, int count,
       MPI_Datatype datatype, int source,
       int tag, MPI_Comm comm, MPI_Request *request
);
```

Once the user started the non-blocking operation the user can wait for the completion using the `MPI_Wait` with a pointer to the request, and a pointer which will be used to set the status of the completion.

There is another option to check if the operation is completed without blocking the execution, the `MPI_Test` primitive takes a pointer to the request, a pointer to a flag, and a pointer to a `MPI_Status` object. If the operation is not yet completed it will set the flag to 0, if it is completed it will set the flag to 1, and the status to the completion status.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

2.1.3 Collective Communication

MPI also offers collective operations, operations that involve all the processes, more exactly all the processes in an MPI communicator. These collective operations can be used to perform synchronization between processes, exchange data one to all or all to one, and more advanced operations that require all the processes to participate.

To synchronize the processes the user can use the `MPI_Barrier` primitive with the `MPI_Comm` of the ranks that have to synchronize. All the ranks will block their execution (at least at a thread level) until all other ranks reach the barrier.

To send data to all other ranks, the user can use the `MPI_Bcast` primitive with a pointer to the data, the number of elements, the data type, the sender rank id, and the MPI communicator.

The user can use more advanced communication patterns, such as the `MPI_Allreduce` primitive, where every process shares its data and then apply an operation. The parameters of this primitive are a pointer to the sending data, a pointer to the receiving data, the number of elements to send, the data type, the operation to apply, and the MPI communicator.

```
int MPI_Barrier(MPI_Comm comm);
int MPI_Bcast(
       void *buf, int count, MPI_Datatype datatype,
       int root, MPI_Comm comm
);
int MPI_Allreduce(
       void *sendbuf, void *recvbuf,
       int count, MPI_Datatype datatype,
       MPI_Op op, MPI_Comm comm
);
```

Note that the collective primitives do not have a receive operation, that is because all the involved ranks have to call the same operation.

Collective operations also have their non-blocking versions, which work exactly the same as the non-blocking point-to-point primitives.

2.2 IBVerbs

IBVerbs library or InfiniBand Verbs, is an implementation of Remote Direct Memory Acces (RDMA) verbs for InfiniBand networks. It handles the control path of creating, modifying, querying, and destroying resources such as Protection Domains (PD), Completion Queues (CQ), Queue-Pairs (QP), Shared Receive Queues (SRQ), Memory Regions (MR). It also handles sending and receiving data posted to QPs and SRQs, getting completions from CQs using polling and completions events.

The control path is implemented through system calls to the uverbs kernel module which further calls the low-level HW driver. The data path is implemented through calls made to low-level HW library which, in most cases, interacts directly with the HW providers kernel and network stack bypass (saving context/mode switches) along with zero copy and an asynchronous I/O model.

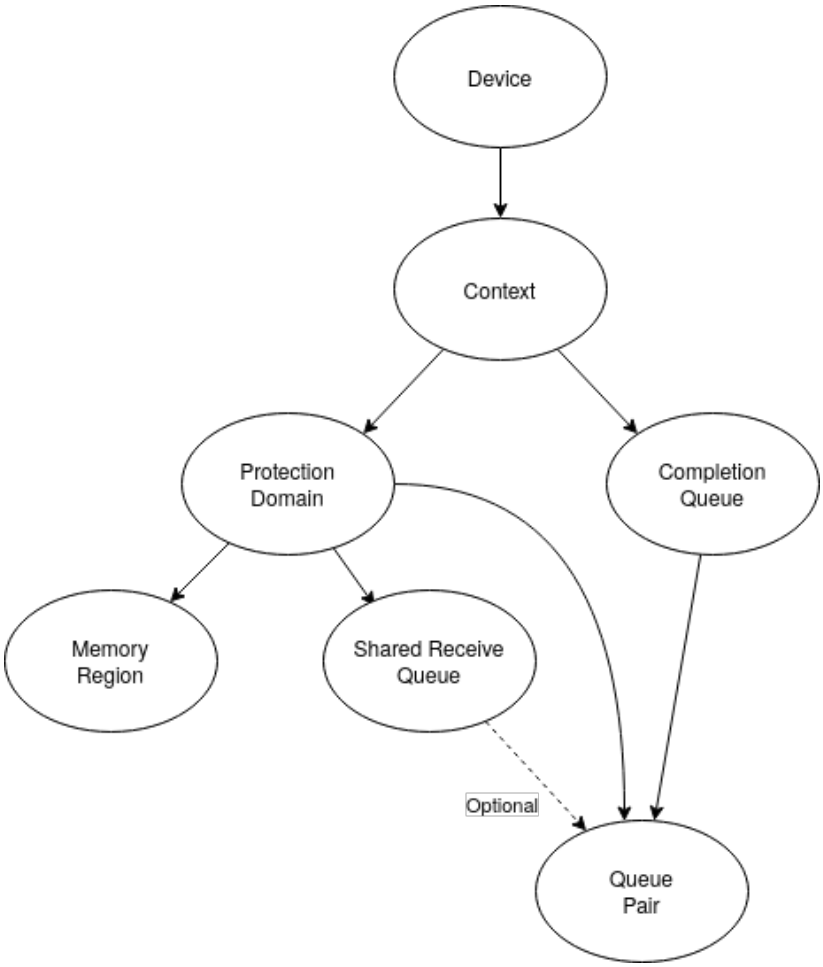


Fig. 2.1: Diagram of the main objects of IBverbs, and its dependencies. The straight arrows are mandatory dependencies, and the dotted arrows are optional dependencies.

To start any IBVerbs application, the user needs to search for devices in the current machine, usually choose one of them, if there are more than one, and get the context of that device. These steps are mandatory, as all the other structures of IBVerbs are dependent on the device context.

With the context of the device, the user can query the attributes of the device and the status of the ports. It can also create a Protection Domain and Completion Queues.

From the protection domain, the user can create Memory Regions, which will allow to write/read the memory from remote processes, and a Shared Receive Queue, which allows sharing Receive Work Requests making it easier and more flexible to manage resources.

Once the user has a Protection Domain, Completion Queues for local and remote completions (can be the same queue), and optionally a Shared Receive Queue, the user can start creating Queue Pairs for each pair of processes, allowing the communication between two processes.

Finally, to communicate, the process that starts the operation has to post a Send Work Request to a Queue Pair, and the process that has to receive this operation, has to post a Receive Work Request to the Queue Pair, or the Shared Receive Queue. The completion of these operations will generate a completion that can be processed by polling the Completion Queue.

2.2.1 Device

To get the list of devices, the user has to use the `ibv_get_device_list` primitive with a pointer to the number of devices. The primitive will return a NULL-terminated array of RDMA devices and will set the number of devices if the pointer is not NULL. If the call did not find any device the number of devices is set to 0 and non-NULL is returned. If the call fails, returns NULL and sets the number of devices with the error number (errno).

The array should be released with `ibv_free_device_list`.

```
struct ibv_device** ibv_get_device_list(int *num_devices);
void ibv_free_device_list(struct ibv_device **list);
```

The user can get the device name or the device GUID, using the `ibv_get_device_name` primitive or the `ibv_get_device_guid` primitive, respectively, with a pointer to the device. These primitives are useful in case the user wants to search for a specific device.

```
const char *ibv_get_device_name(struct ibv_device *device);
uint64_t ibv_get_device_guid(struct ibv_device *device);
```

2.2.2 Context

With a device selected, the user can create a context for the device, using the `ibv_open_device` primitive with a pointer to the device. This context is essential to create all the other IBVerbs structures and should be released only using `ibv_close_device` once the device is not used anymore.

```
struct ibv_context *ibv_open_device(struct ibv_device *device);
int ibv_close_device(struct ibv_context *context);
```

2.2.3 Queries

With the device context, the user can query for all the device attributes using the `ibv_query_device` primitive with the context and a pointer to a (ibv_device_attr) structure. The call will set the `ibv_device_attr` with the attributes of the device.

If the user wants information on a port of the device, can use the `ibv_query_port` primitive with the context, a port number, and a pointer with the attributes of the port. The call will set the pointer to the attributes with the current attributes of the port.

```
int ibv_query_device(struct ibv_context *context, struct ibv_device_attr *
    device_attr);
int ibv_query_port(
        struct ibv_context *context, uint8_t port_num,
        struct ibv_port_attr *port_attr
);
```

2.2.4 Protection Domain

The protection domain is used to create a group of objects that can work together, an application can have multiple protection domains, but the objects of the domain cannot interact between domains, or it will end up with completion errors.

To create the protection domain, the user has to use the `ibv_alloc_pd` primitive with the device context. The call will return the domain if it is allocated and created the protection domain, and NULL if it failed.

To deallocate the protection domain, the user has to use the `ibv_dealloc_pd` primitive with a pointer to the protection domain.

```
struct ibv_pd *ibv_alloc_pd(struct ibv_context *context);
int ibv_dealloc_pd(struct ibv_pd *pd);
```

2.2.5 Memory Region

A memory region is used to grant remote access to a section of the local memory, the memory has to be previously allocated.

To register a memory region, the user has to use the `ibv_reg_mr` primitive, with a pointer to the protection domain, a pointer to the memory, the length of the memory region, and the access flags. The call will return a pointer to the memory region object.

To deregister the memory region, the user has to use the `ibv_dereg_mr` primitive with a pointer to the memory region.

```
struct ibv_mr *ibv_reg_mr(
       struct ibv_pd *pd, void *addr,
       size_t length, enum ibv_access_flags access
);
int ibv_dereg_mr(struct ibv_mr *mr);
```

2.2.6 Completion Queue

The completion queues are used to register the completion of a Work Request that was previously posted in a Work Queue, this completion indicates the Work Request has finished and contains the information of how had been completed (e.g. error). A completion queue can be shared between multiple Queue Pairs, and accept completions for sending, receiving, or both.

To create a completion queue the user has to use the `ibv_create_cq` primitive with the device context and the minimum capacity of the completion queue. Optionally, the user can add a `cq_context` object, a channel, and an MSI-X completion vector. This primitive returns a pointer to the already initialized completion queue.

The size of the completion queue can be changed at any point of the application using the `ibv_resize_cq` primitive with a pointer to the completion queue and the new size.

The user can destroy the completion queue using the `ibv_destroy_cq` primitive with a pointer to the completion queue.

```
struct ibv_cq *ibv_create_cq(
       struct ibv_context *context, int cqe,
       void *cq_context, struct ibv_comp_channel *channel,
       int comp_vector
);
int ibv_resize_cq(struct ibv_cq *cq, int cqe);
int ibv_destroy_cq(struct ibv_cq *cq);
```

To get the completions, the user has to use the `ibv_poll_cq` primitive with a pointer to the completion queue, the maximum number of completions that want to poll, and an array of completions to be filled. This array must have at least the length of the maximum number of completions that can poll. The call will return the number of polled Work Completions and will fill the completion array with the Work Completions that have been polled.

```
struct ibv_wc {
        uint64_t  wr_id;
        enum ibv_wc_status status;
        enum ibv_wc_opcode opcode;
        uint32_t  vendor_err;
        uint32_t  byte_len;
        uint32_t  imm_data;
        uint32_t  qp_num;
        uint32_t  src_qp;
        int   wc_flags;
        uint16_t  pkey_index;
        uint16_t  slid;
        uint8_t   sl;
        uint8_t   dlid_path_bits;
};

int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc);
```

2.2.7 Shared Receive Queue

The Shared Receive Queue (SRQ) is a Receive Queue that can be shared between Queue Pairs, making it easier and more flexible to post Receive Work Requests for different Queue Pairs.

To create a Shared Receive Queue the user has to use the `ibv_create_srq` primitive with a pointer to the protection domain, and a pointer to a `ibv_srq_init_attr` structure that contains the attributes of the SRQ. The call will return a pointer to the SRQ object.

To destroy the Shared Receive Queue the user has to use the `ibv_destroy_srq` primitive with a pointer to the SRQ object.

```
struct ibv_srq *ibv_create_srq(
        struct ibv_pd *pd,
        struct ibv_srq_init_attr *srq_init_attr
);
int ibv_destroy_srq(struct ibv_srq *srq);
```

2.2.8 Queue Pair

The Queue Pairs (QP) are the connections between the two processes. The user can post Work Requests to a Queue Pair to communicate, and these Work Requests will generate a Work Completion in the Completion Queue linked to the Queue Pair.

To create a Queue Pair the user has to use the `ibv_create_qp` primitive with a pointer to the Protection Domain and a pointer to a `ibv_qp_init_attr` struct, with the necessary attributes and objects to create the Queue Pair. The call will return a pointer to the QP object.

To destroy the Queue Pair, the user has to use the `ibv_destroy_qp` primitive with a pointer to the QP object.

```
struct ibv_qp_init_attr {
        void  *qp_context;
        struct ibv_cq *send_cq;
        struct ibv_cq *recv_cq;
        struct ibv_srq *srq;
        struct ibv_qp_cap cap;
        enum ibv_qp_type qp_type;
        int   sq_sig_all;
};

struct ibv_qp *ibv_create_qp(
        struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr
);
int ibv_destroy_qp(struct ibv_qp *qp);
```

2.2.9 Post Operations

To communicate between processes, the user has to post a Send Work Request in the Queue Pair of the sending process and post a Receive Work Request in the same Queue Pair of the receiving process. These Work Requests will generate a Work Completion with the completion information and will post them in the corresponding Completion Queue.

To post a Send Working Request the user has to use the `ibv_post_send` primitive with a pointer to the Queue Pair, a pointer to the, already initialized, Send Work Request, and a pointer to an error Send Work Request.

```
struct ibv_send_wr {
        uint64_t  wr_id;
        struct ibv_send_wr *next;
        struct ibv_sge *sg_list;
        int   num_sge;
        enum ibv_wr_opcode opcode;
        int   send_flags;
        uint32_t  imm_data;
        union {
                struct {
                        uint64_t remote_addr;
                        uint32_t rkey;
                } rdma;
                struct {
                        uint64_t remote_addr;
                        uint64_t compare_add;
                        uint64_t swap;
                        uint32_t rkey;
                } atomic;
                struct {
                        struct ibv_ah *ah;
                        uint32_t remote_qpn;
                        uint32_t remote_qkey;
                } ud;
        } wr;
};

int ibv_post_send(
        struct ibv_qp *qp, struct ibv_send_wr *wr,
        struct ibv_send_wr **bad_wr
);
```

To post a Receive Work Request the user has to use the `ibv_post_recv` primitive with a pointer to the Queue Pair, a pointer to the already initialized Receive Work Request, and a pointer to an error Receive Work Request.

Optionally, if there is a Shared Receive Queue linked to the Queue Pair, the user post a Receive Work Request to the Shared Receive Queue using the `ibv_post_srq_recv` primitive with a pointer to the Shared Receive Queue, a pointer to the already initialized Receive Work Request, and a pointer to an error Receive Work Request.

```
struct ibv_recv_wr {
       uint64_t  wr_id;
       struct ibv_recv_wr *next;
       struct ibv_sge *sg_list;
       int   num_sge;
};


int ibv_post_recv(
       struct ibv_qp *qp, struct ibv_recv_wr *wr,
       struct ibv_recv_wr **bad_wr
);
int ibv_post_srq_recv(
       struct ibv_srq *srq,
       struct ibv_recv_wr *recv_wr,
       struct ibv_recv_wr **bad_recv_wr
);
```

2.3 Lightweight Parallel Foundations

Lightweight Parallel Foundations (LPF) [8] is an interoperable and model-compliant communication layer adhering to a strict performance model of parallel computations. The principles of interoperability and model compliance suffice for the practical use of immortal algorithms: algorithms that are proven optimal once, and valid forever. These are ideally also implemented once, and usable from a wide range of sequential and parallel environments.

LPF follows a Single Program Multiple Data (SPMD) and Bulk Synchronous Parallel (BSP) paradigms, where the main communication primitives leverage remote direct memory access (RDMA). LPF consists of twelve primitives, each with strict performance guarantees and two of which enable interoperability.

lpf_exec: $\mathcal{O}(Ng + l)$	**lpf_put:** $\Theta(1)$
lpf_hook: $\mathcal{O}(Ng + l)$	**lpf_get:** $\Theta(1)$
lpf_rehook: $\mathcal{O}(Ng + l)$	**lpf_sync:** $hg + l$
lpf_register_local: $\mathcal{O}(M + N)$	**lpf_deregister:** $\Theta(1)$
lpf_register_global: $\mathcal{O}(M + N)$	**lpf_probe:** $\Omega(1)$
lpf_resize_memory_register: $\mathcal{O}(N)$	**lpf_resize_message_queue:** $\mathcal{O}(N)$

Fig. 2.2: All LPF primitives and their asymptotic run-time costs. M depends on the LPF state, N depends on function arguments, and h, g, l are system constants that depend on the number of processes, number of processes per node, and the modality (Shared-memory, RDMA, etc).

The algorithmic complexity analysis from Figure 2.2 corresponds to the run-time of its implementation, and as the sequential algorithms, it should transfer to different architectures without invalidating its complexity analysis. The parallel case differs from the sequential only in communication between processing units.

2.3.1 Initialization

The first step in an LPF application is to initialize the process and interconnect them, to do so LPF offers two methods. The first method is using the `lpf_exec` with the `LPF_ROOT` context, the number of processes, the SPMD function, and the arguments.

```
lpf_err_t lpf_exec(
       lpf_t ctx, lpf_pid_t P,
       lpf_spmd_t spmd, lpf_args_t args
);
```

The other method consists in initialize a `lpf_init_t` object with TCP/IP or an MPI communicator, using the `lpf_mi_initialize_over_tcp` primitive or `lpf_mpi_initialize_with_mpicomm` primitive respectively. Once the `lpf_init_t` object is created we can start the SPMD function using the `lpf_hook` primitive.

```
lpf_err_t lpf_mpi_initialize_over_tcp(
       const char * server, const char * port, int timeout,
       lpf_pid_t pid, lpf_pid_t nprocs,
       lpf_init_t * init
);
lpf_err_t lpf_mpi_initialize_with_mpicomm(MPI_Comm comm, lpf_init_t *init);
lpf_err_t lpf_hook(lpf_init_t init, lpf_spmd_t spmd, lpf_args_t args);
```

2.3.2 Message Queue

Before being able to communicate between processes the user has to register a Message Queue to post the communication operations before executing them at the synchronization phase. The `lpf_resize_message_queue` primitive is used to register or resize this message queue, with the LPF context and the maximum expected messages that the process will post between synchronizations.

```
lpf_err_t lpf_resize_message_queue(lpf_t ctx, size_t max_msgs);
```

2.3.3 Memory Registration

To be able to register memory regions, the user has to define the maximum number of memory slots, that will be used to register the memory regions, using the `lpf_resize_memory_register` with the LPF context and the maximum number of memory regions that will be registered.

```
lpf_err_t lpf_resize_memory_register(lpf_t ctx, size_t max_regs);
```

Once determined the number of memory slots the user can register the memory regions using two different methods, `local` for inter-node communication using the communication primitives, and `global` for intra-node communication.

To register a local memory slot, the user has to use the `lpf_register_local` primitive, with the LPF context, the pointer to the beginning of the memory region, the size of this memory region, and a pointer to a memory slot object.

```
lpf_err_t lpf_register_local(
       lpf_t ctx, void * pointer, size_t size,
       lpf_memslot_t * memslot
);
```

To register a global memory slot, the user has to use the `lpf_register_global` primitive, with the same parameters as the `lpf_register_local` primitive.

```
lpf_err_t lpf_register_global(
       lpf_t ctx, void * pointer, size_t size,
       lpf_memslot_t * memslot
);
```

To deregister a memory slot, usually at the end of the application, the user has to use the `lpf_deregister` primitive, with the LPF context and the memory slot object.

```
lpf_err_t lpf_deregister(lpf_t ctx, lpf_memslot_t memslot);
```

2.3.4 Communication Primitives

Once the user has finished with the setup of the application and can start the core of the BSP application, the computation phase does not require any intervention from LPF. For the communication phase, LPF offers to the user two primitives, `lpf_put` and `lpf_get`. Both primitives do not issue the communication directly, instead, only post the information needed to issue the communication to the message queue and the communication itself is done in the synchronization primitive.

The `lpf_put` primitive is used to write from your local memory slot to a remote memory slot. This primitive uses the LPF context, the source local memory slot and its offset, the destination process id, the destination memory slot, and its offset, the size of the memory segment to write, and an `lpf_msg_attr_t` object. In the actual implementation of LPF the `lpf_msg_attr_t` object is unused.

```
lpf_err_t lpf_put(
       lpf_t ctx, lpf_memslot_t src_slot,
       size_t src_offset, lpf_pid_t dst_pid,
       lpf_memslot_t dst_slot, size_t dst_offset,
       size_t size, lpf_msg_attr_t attr
);
```

The `lpf_get` primitive is used to read from a remote memory slot and store the result in a local memory slot. This primitive uses the LPF context, the source process id, the source remote memory slot and its offset, the destination local memory slot and its offset, the size of the memory region to read, and an `lpf_msg_attr_t` object. In the actual implementation of LPF the `lpf_msg_attr_t` object is unused.

```
lpf_err_t lpf_get(
       lpf_t ctx, lpf_pid_t src_pid,
       lpf_memslot_t src_slot, size_t src_offset,
       lpf_memslot_t dst_slot, size_t dst_offset,
       size_t size, lpf_msg_attr_t attr
);
```

2.3.5 Synchronization Primitive

The synchronization phase of LPF is done by using the `lpf_sync` primitive. In LPF the synchronization also does the communication phase as the communication primitives `lpf_put` and `lpf_get` do not issue the communication directly as mentioned in the previous section. The `lpf_sync` implementation is divided into four phases: a global barrier and a first meta-data exchange informing the other processes of the data destinations of each `lpf_put` and `lpf_get`; a write-conflict resolution on the data destination followed by a second meta-data exchange informing the data sources what data can be sent without overlap; the actual data exchange and waiting for its completion; a final barrier that ensures all communication has finished before starting the next iteration of the BSP model.

The `lpf_sync` primitive has the LPF context and an `lpf_sync_attr_t` object as parameters. In the actual implementation of LPF the `lpf_sync_attr_t` object is unused.

```
lpf_err_t lpf_sync(lpf_t ctx, lpf_sync_attr_t attr);
```

2.3.6 Example

In this example, we have several processes, where each process has an array of a length equal to the number of processes. This array is initialized with the process id, and using a ring pattern communication with `lpf_get` primitives, we expect to finish the program by having the array with the value of each process id in its position (e.g. in the position 0 of the array it should be the process id of the first process).

To initialize this example we create an MPI environment using `MPI_Init_thread`. Then we create the arguments object and we initialize it with the arguments of the program. We create the `lpf_init_t` object using the `lpf_mpi_initialize_with_mpicomm` primitive with the `MPI_COMM_WORLD` which we previously initialized with `MPI_Init_thread`. Once we have initialized all the necessary objects, we call out the application with `lpf_hook`. Finally, once our application has finished, we call `lpf_mpi_finalize` to finalize the `lpf_init_t` object, and `MPI_Finalize` to finalize MPI.

```
1
2  void application(
3         lpf_t ctx, lpf_pid_t pid,
4         lpf_pid_t nprocs, lpf_args_t args
5  );
6
7  int main(int argc, char** argv)
8  {
9       const int required = MPI_THREAD_SERIALIZED;
10
11        int provided;
12        MPI_Init_thread(&argc, &argv, required, &provided);
13        if (provided != required) {
14               fprintf(stderr, "Error: MPI threading level not supported!\n")
                     ;
15               return 1;
16        }
17
18        lpf_args_t args;
19        memset(&args, 0, sizeof(lpf_args_t));
20        args.input = argv; args.input_size = argc;
21
22        lpf_init_t init;
23        lpf_mpi_initialize_with_mpicomm(MPI_COMM_WORLD, &init);
24
25        lpf_hook(init, &application, args);
26
27        lpf_mpi_finalize(init);
28        MPI_Finalize();
29
30 }
```

Code 2.1: Example of a LPF application initialization from a MPI Communicator

Inside the application, we get the arguments, which in this application are unused, and we initialize the array of process ids. We do the initialization of the message queue with `lpf_resize_message_queue` with a size equal to one, as we only have one get between synchronizations, and the initialization of the memory slots, in this case, one, as we only need one memory region. After that, we synchronize the process to make sure that all the processes have been initialized correctly, and we register the array as a global memory region with `lpf_register_global`, and again we synchronize to make sure all processes have registered the array correctly. Once all are initialized we can start with the main function, in this application there is not a computation phase, so we start directly with the communication calling `lpf_get` to get the next process id, then we synchronize all the process to make sure that all the arrays are updated, and we repeat for the number of processes. Finally, before finishing the application, we need to deregister the memory region by using `lpf_deregister`.

```
void application(
        lpf_t ctx, lpf_pid_t pid,
        lpf_pid_t nprocs, lpf_args_t args
) {
        char **argv = (char **) args.input;
        int argc = args.input_size;
        int err;

        lpf_t lpf = ctx;
        int rank = pid;
        int nranks = nprocs;

        int A[nranks];
        for(int i = 0; i < nranks; i++) A[i] = rank;
        for(int i = 0; i < nranks; i++) printf("%d ", A[i]);
        printf("\n");

        lpf_memslot_t LOCAL;

        lpf_resize_message_queue(lpf, 1);
        lpf_resize_memory_register(lpf, 1);

        lpf_sync(lpf, LPF_SYNC_DEFAULT);

        lpf_register_global(lpf, A, nranks*sizeof(int), &LOCAL);

        lpf_sync(lpf, LPF_SYNC_DEFAULT);

        int src = (rank + 1) % nranks;
        for (int i = 1; i < nranks; i++){
            int idx = (rank + i)%nranks;
            lpf_get(lpf, src, LOCAL, idx*sizeof(int), LOCAL, idx*sizeof(int),
                sizeof(int), LPF_MSG_DEFAULT);
               lpf_sync(lpf, LPF_SYNC_DEFAULT);
        }

        for(int i = 0; i < nranks; i++) printf("%d ", A[i]);
        printf("\n");

        lpf_deregister(lpf, LOCAL);

}
```

Code 2.2: Example of a LPF application.

2.4 OmpSs-2 Task-Based Programming Model

OmpSs-2 [7], developed at the Barcelona Supercomputing Center (BSC), is the second generation of the OmpSs programming model. It combines elements from both the OpenMP and StarSs models, incorporating their design principles to form the foundation of the OmpSs philosophy.

From OpenMP, OmpSs inherits the concept of annotating compiler directives in the source code to facilitate the parallelization of sequential programs. These annotations, while not affecting the program's semantics explicitly, enable the compiler to generate a parallel version of the code. This approach allows users to incrementally parallelize their applications by adding OmpSs directives to different sections of the code.

OmpSs-2 is a programming model that extends the tasking capabilities of OmpSs and OpenMP, providing directives and library routines to develop concurrent applications using high-level programming languages such as C, C++, and Fortran. It introduces support for task nesting and fine-grained dependencies across different nesting levels.

The primary objective of OmpSs-2 is to offer a productive environment for developing parallel applications on modern HPC systems. This entails providing competitive performance while prioritizing ease of use for all users. Moreover, OmpSs-2 aims to extend the OpenMP standard by introducing new directives, clauses, and API services.

In the OmpSs-2 programming model, tasks serve as the fundamental units of work and can be executed asynchronously. To manage the data flow in applications, OmpSs-2 incorporates a robust dependency model. Users annotate tasks with dependencies to specify the data dependencies, and this information is analyzed at runtime to detect potential data races that may arise from the parallel execution of tasks.

The OmpSs-2 programming model encompasses several key functionalities, including:

- Task data environment lifetime: A task is considered complete when its code execution reaches the last statement. Once all child tasks of a task are deeply completed, the task itself becomes deeply completed. The task's data environment, which includes variables captured during task creation, remains preserved until it is deeply completed. However, note that the stack is lost once the task completes.
- Nested dependency domain connection: Dependencies incoming to a task are propagated to its children tasks. When a task finishes, its outgoing dependencies are replaced by the dependencies generated by its children.
- Early release of dependencies: Upon completion of a task, any dependencies not required by any uncompleted child tasks are released. Additionally, specifying a wait clause in a task declaration causes all dependencies to be released once the task is deeply completed.

- Weak dependencies: The weakin/weakout clauses specify potential dependencies that are only necessary for children tasks. As a result, these annotations do not delay the execution of the task.
- Native offload API: An asynchronous API that enables the execution of OmpSs-2 kernels on a specific set of CPUs from any type of application. This allows for efficient offloading of computation to designated CPUs.

These functionalities collectively contribute to the versatility and effectiveness of the OmpSs-2 programming model.

2.4.1 Influencing OpenMP

Over the years, the OmpSs programming model has made significant contributions to the evolution of the OpenMP programming model. Figure 2.3 provides an overview of the key advancements introduced in each version of the OpenMP standard. These advancements include the incorporation of asynchronous tasks, task dependencies, taskloops, and task priorities, among others.

Furthermore, upcoming versions of OpenMP are set to introduce additional features such as multi-dependencies, task reductions, and commutative dependencies, among other advancements. These new features will further enhance the capabilities and flexibility of the OpenMP programming model, allowing developers to efficiently parallelize their code and harness the potential of modern computing systems.

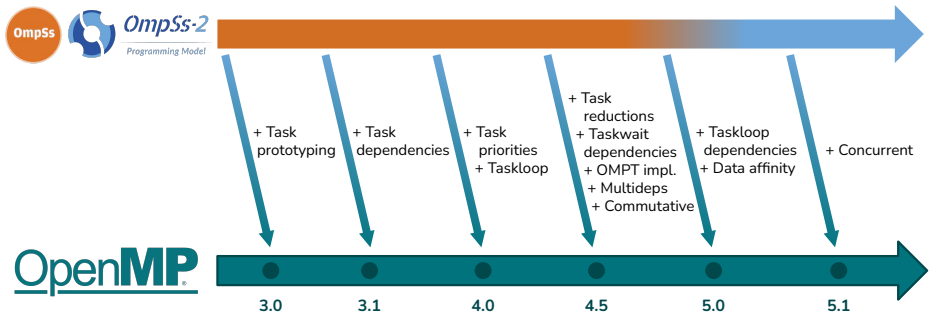


Fig. 2.3: OmpSs and OmpSs-2 features introduced into the OpenMP programming model

2.4.2 Annotating Programs

In this section, we will explain the process of annotating programs to enable the functionalities of OmpSs-2. To maintain simplicity, we will focus on showcasing the annotations for C/C++ programs. However, detailed information regarding Fortran annotations can be found in the OmpSs-2 Specification.

OmpSs-2 annotations must adhere to a specific format. Directives are indicated by the sentinel "oss" followed by the directive name and any applicable clauses. The format of these annotations is illustrated below:

```
#pragma oss directive-name [clauses]
```

By following this format, developers can effectively apply OmpSs-2 directives to their code, enabling the utilization of the OmpSs-2 programming model and its associated functionalities.

2.4.3 Execution Models

A significant distinction between OpenMP and OmpSs-2 is their approach to parallelism. Unlike OpenMP, OmpSs-2 does not require the explicit use of a parallel directive in the user application. OpenMP follows a fork-join execution model, where users must specify the parallel region using this directive to delineate the beginning and end of parallelism. In contrast, OmpSs-2 implements a thread-pool execution model, where parallelism is automatically created at the start of the application. The underlying runtime system manages a pool of threads, which are utilized to execute user tasks. However, users do not have direct control over the thread pool, as it is efficiently managed by the runtime system.

During execution, the OmpSs-2 runtime system creates an initial pool of worker threads without a master thread. The main function is enclosed in an implicit task called the main task, which is added to the queue of ready tasks. When a worker thread retrieves this task from the queue, it begins its execution. Meanwhile, other threads wait for additional ready tasks that can be instantiated by the main task or other running tasks.

Multiple threads execute tasks explicitly or implicitly defined by OmpSs-2 directives. The subsequent sections will delve into the functionalities of creating tasks, OmpSs-2 loops, and explicit synchronization points, providing a comprehensive understanding of these features.

OmpSs-2 utilizes tasks to express parallelism, representing independent code segments executed by parallel resources. When a task declaration is reached, an instance of the task is created and delegated to the OmpSs-2 runtime system. The runtime system analyzes task dependencies and determines the correct execution order. Once all dependencies are satisfied, the task becomes ready for execution. The execution may occur immediately or be deferred based on scheduling constraints and resource availability. Additionally, tasks can block, allowing other ready tasks to utilize available resources. OmpSs-2 optimizes parallel resource utilization and dynamic task scheduling.

The users can create a task using:

```
#pragma oss task [clauses]
{*code*}
```

OmpSs-2 tasks support multiple clauses, including variable data-sharing. The private clause privatizes a variable with an uninitialized value. The firstprivate clause privatizes and initializes the variable with its original value. The shared clause allows tasks to work with the original variable.

For managing dependencies, OmpSs-2 enables the addition of input, output, and input-output dependencies using clauses: in(variable-list), out(variable-list), and inout(variable-list) respectively. Section 2.4.4 provides detailed explanations of dependencies. Other clauses like priority, final, and if can be specified as well.

In addition to basic task functionality, OmpSs-2 offers more advanced approaches such as task loops and nesting. While these techniques provide powerful capabilities, for brevity, we won't delve into their details here. You can refer to the OmpSs-2 Specification for comprehensive information on these advanced features.

OmpSs-2 supports explicit synchronization points through the taskwait directive. However, it is recommended to minimize its usage as it significantly limits parallelism within the calling parallel context. The OmpSs-2 dependency model offers alternative techniques to avoid the need for taskwait, which are detailed in Section 2.4.4.

2.4.4 Dependency Model

In OmpSs-2, asynchronous parallelism is achieved through data dependencies among tasks. Tasks often require data to perform computations and can generate new data that other tasks may need as input.

During the execution of an OmpSs-2 program, the runtime system analyzes the data dependencies and creation order of tasks. This information is used to establish execution-order constraints, known as task dependencies, ensuring the correct sequence of task execution.

When a new task is created, its dependencies are compared with those of existing tasks. If dependencies such as Read-after-Write (RaW), Write-after-Write (WaW), or Write-after-Read (WaR) are identified, the task becomes a successor to the corresponding existing tasks. Tasks are scheduled for execution once all their predecessors have completed or immediately upon creation if there are no predecessors.

Dependencies in OmpSs-2 enable the user to specify the data that tasks are waiting for and producing. It is the user's responsibility to accurately define these dependencies for each task. OmpSs-2 supports four fundamental types of dependencies:

- Tasks with "in(variable-list)" dependencies cannot run until all "out", "inout", and concurrent predecessors are finished.

- Tasks with ”out(variable-list)” dependencies cannot run until all ”in”, ”out”, ”in-out”, and concurrent predecessors are finished.
- Tasks with ”inout(variable-list)” dependencies are equivalent to declaring both ”in” and ”out” dependencies.
- Tasks with ”concurrent(variable-list)” dependencies cannot run until all ”in”, ”out”, and ”inout” predecessors are finished. They can run concurrently with concurrent predecessors.

Additionally, OmpSs-2 allows declaring dependencies on memory positions pointed by pointers. If the pointer is NULL, the dependency is ignored. For example, the following code snippet declares an input dependency on the variable ”a”:

```
int a = 0;
int *p = &a
#pragma oss task in(p)
{*code*}
```

OmpSs-2 also provides the capability to declare dependencies on multiple elements of an array in a single expression. The runtime system efficiently manages these dependencies on array regions. There are two ways to define such dependencies, as illustrated in the example below:

- In the first method, all elements of the array within the range [lower, upper] (inclusive) are referenced.
- In the second method, all elements of the array within the range [lower, lower+(size-1)] (inclusive) are referenced.

```
#pragma oss task inout(a[lower:upper]) out(a[lower;size])
```

2.4.5 Task Scheduling

During task execution, the OmpSs-2 runtime system can choose to switch to another eligible task at a scheduling point. Scheduling points can occur within task-generating code, taskyield directives, taskwait directives, or after completing a task.

The set of eligible tasks initially consists of tasks in the ready queue, which have all their dependencies satisfied. Task switching involves starting a task that has not been executed before or resuming the execution of a partially executed task. However, once a task has been executed by a thread, only that same thread can resume its execution.

Furthermore, if a task is created with an if clause evaluates to false, the current task must suspend its execution until the newly created task is completed. Similarly, in a final context, when task-generating code is encountered, the child task is immediately executed right after its creation.

2.4.6 Reference Implementation

The OmpSs-2 reference implementation relies on two key components: Mercurium and *Nanos6* tools.

Mercurium serves as a source-to-source compiler, enabling the transformation of OmpSs-2 directives into a parallelized version of the application. It plays a crucial role in facilitating the integration of OmpSs-2 features into the codebase.

Nanos6, on the other hand, functions as the runtime system that manages parallelism within user applications. It offers a comprehensive suite of services necessary for effective parallel execution. Further details about these tools can be found in Chapter 4, where we provide an in-depth exploration of their functionalities.

3 | Related Work

Task-Aware Light Parallel Foundations is not the first Task-Aware library, there are previous works that explored the idea of combining a data-flow model with a communication API, two of these libraries are Task-Aware MPI (TAMPI) and Task-Aware GASPI (TAGASPI), in this chapter we will introduce both libraries.

3.1 Task-Aware MPI (TAMPI)

The MPI and OpenMP standards were not designed to perform MPI communications from multiple OpenMP tasks concurrently [9, 10]. The Task-Aware MPI (TAMPI) library [10, 13] overcomes this limitation by enabling the safe and efficient taskification, and thus parallelization, of MPI communications. The library allows exploiting the parallelism of communications, relaxing their execution order restrictions, and overlapping of computation and communication tasks. The communication tasks can use either blocking or non-blocking MPI operations, including point-to-point and collectives.

The TAMPI library provides two main modes: blocking and non-blocking. The blocking mode provides support for tasks that run blocking MPI operations (e.g., `MPI_Recv`), whereas the non-blocking mode provides support for tasks that issue non-blocking ones (e.g., `MPI_Irecv`). On the one hand, the blocking mode makes all MPI blocking calls from the user application to pause the running task and transparently re-uses the compute resource for running other ready tasks while the communications progress in the background. The TAMPI library coordinates with the tasking runtime system to prevent wasting compute resources by avoding busy-waiting for ongoing communications. Eventually, paused task are resumed once its communications complete.

On the other hand, the non-blocking mode provides an efficient mechanism focused on tasks that issue asynchronous MPI operations. Once again, the idea is to exploit concurrent communications through multiple tasks. To this end, the library provides a new asynchronous and non-blocking function, called `TAMPI_Iwait`, that has the same parameters as the `MPI_Wait`. A task may call a non-blocking MPI operation (e.g., `MPI_Irecv`) and generate an MPI request for checking its progress. Instead of manually checking the progress, the task can call `TAMPI_Iwait` and bind the task completion to the finalization of the MPI request passed as a parameter. Due to its asynchronous nature, the function returns immediately without specifying whether the operation

already finalized or not. The calling task continues but will not complete until (1) it finishes its execution and (2) all its bound MPI operations finalize.

In OpenMP and OmpSs-2, the data dependencies of a task are released just after the task completes. The `TAMPI_Iwait` relies on the fact that the user annotates the communication tasks with the send buffers as input dependencies and the receive buffers as output dependencies. When one of these tasks completes, its dependencies on the communication buffers are released, and its successor tasks become ready to consume/reuse them. An example of a task receiving data with TAMPI is shown below:

```
#pragma omp depend(out: buffer[0:N])
{
      MPI_Request request;
      MPI_Irecv(buffer, N, MPI_INT, src, tag, MPI_COMM_WORLD, &request);
      TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
      // Do not consume the buffer here, may not be received yet
}

#pragma omp depend(in: buffer[0:N])
consumeBuffer(buffer, N);
```

Code 3.1: OpenMP tasks receiving and consuming data thanks to the non-blocking mechanism of TAMPI.

For operating in both modes, the library instantiates a hidden task that periodically checks the progress of communication requests and notifies the tasking runtime system when a task has finished all the pending operations. The library leverages all the tasking runtime API functions explained above.

3.2 Task-Aware GASPI (TAGASPI)

The Task-Aware MPI library provides task-awareness to MPI point-to-point and collective operations. However, it does not include support for one-sided operations, also known as remote memory access (RMA). One-sided operations are those where the issuing process directly access the local memory of another process with no intervention from the remote side. These operations can leverage the RDMA hardware support that is present in most modern network fabrics. One-sided models minimize the overheads and complexities of two-sided interface implementations by removing the message tag matching, the message queuing, the internal data buffering, and their complex locking schemes.

The Task-Aware GASPI (TAGASPI) [11, 12] library provides task-awareness features for one-sided operations. GASPI is a simple RMA interface that provides efficient one-sided operations with very fine-grained synchronization primitives. It provides functions to read and write from/to another process, send lightweight remote notifications, and wait for these notifications. The idea behind TAGASPI is to provide the task-aware variants of these functions and allowing tasks to issue one-sided operations and asynchronously wait for remote notifications.

All the functions offered by TAGASPI are non-blocking and asynchronous and behave similarly to the TAMPI non-blocking mode's services. The communication tasks leveraging TAGASPI will complete once they finish executing and all their operations have completed. In this case, however, there are extra complexities associated with the one-sided models. Some of these are that the remote writer must know exactly where to write the data on the remote side, and that notifications may be needed to acknowledge when a buffer can be written safely without overwriting previous values.

4 | Tools & Methodology

In this chapter, we will provide an overview of the tools and methodology employed in this project. This section aims to familiarize readers with the key components utilized and the approach undertaken throughout the study.

4.1 Tools

In the following sections, we will introduce the tools and implementations used in this project. We utilized the MPICH implementation for MPI, *Nanos6* as the OmpSs-2 backend, and the Mercurium compiler for program compilation.

For an in-depth analysis of program execution, we employed ovni and Paraver. Ovni allowed us to extract execution information and generate traces, while Paraver facilitated the visualization of these traces, providing valuable insights into program behavior.

4.1.1 Mercurium

Mercurium [14], developed at the BSC, is a source-to-source compilation infrastructure supporting C, C++, and Fortran languages. It is primarily used in the Nanos environment for OpenMP implementation, but it can also be extended for other programming models and compiler transformations such as Cell Superscalar (CellSs) and Software Transactional Memory.

Designed with a plugin-based architecture, Mercurium consists of dynamically loaded C++ plugins representing different compiler phases. Code transformations can be implemented directly on the source code without the need for internal syntactic representation modifications.

Mercurium facilitates the OmpSs programming models by processing OmpSs compiler directives and converting them into runtime library function calls. It can generate platform-specific code for various target devices like CPUs, GPUs, and FPGAs.

Figure 4.1 illustrates the workflow of Mercurium for OmpSs-2. The user provides directive-based source code, which is processed by Mercurium. The directives are

transformed into corresponding function calls of the Nanos runtime system. The transformed code is then compiled using the chosen native compiler (e.g., gcc) and linked with the Nanos runtime system's library.

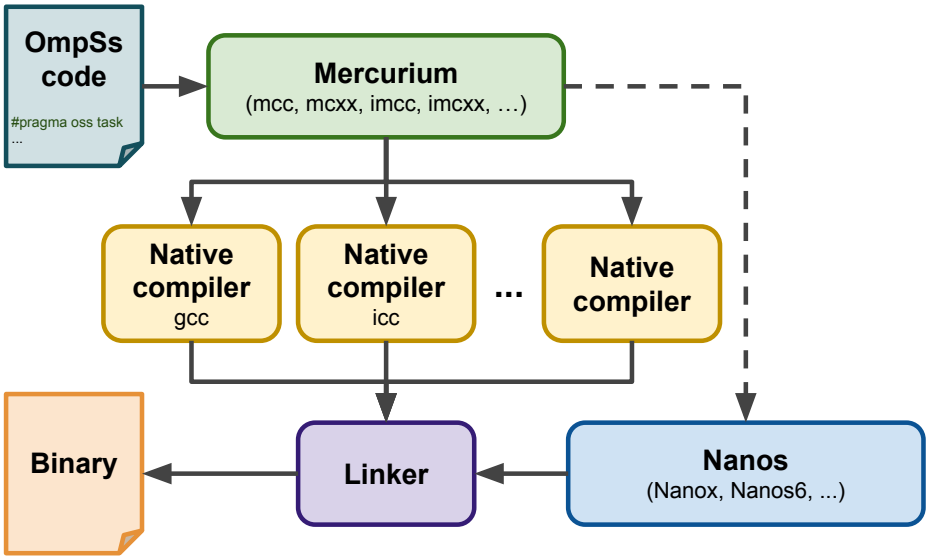


Fig. 4.1: Mercurium workflow for OmpSs-2 source codes.

4.1.2 Nanos6

Nanos6 [15], developed at the BSC, is a powerful runtime system library that manages parallelism in user applications. It handles task scheduling, execution, and maintains the necessary task dependency constraints.

As the successor to the Nanox runtime system, Nanos6 fully supports all the constructs of the OmpSs-2 programming model. It offers a range of services including task creation, synchronization, data movement, various task scheduling policies, and support for heterogeneous resources. The structure of Nanos6 is depicted in Figure 4.2.

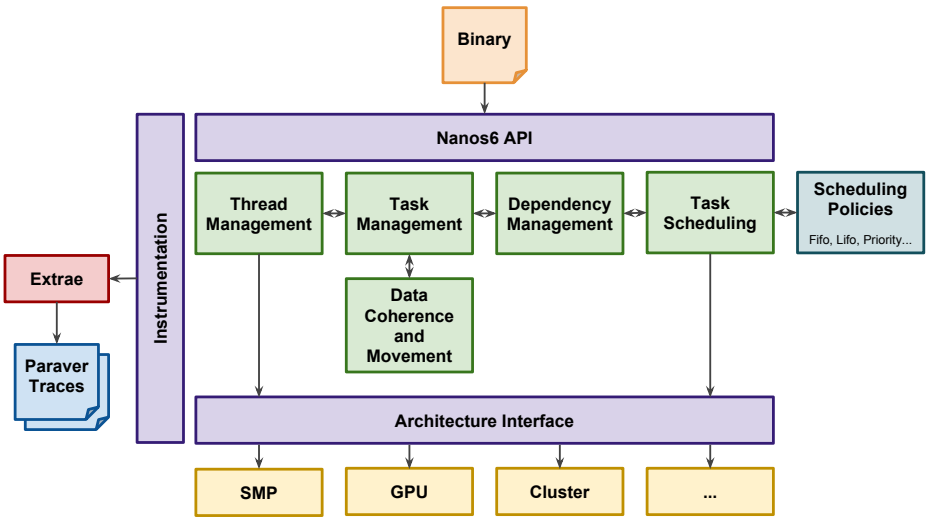


Fig. 4.2: Nanos6 runtime system structure.

Nanos6 provides a well-defined API that can be accessed from any user application. However, it is highly recommended to use it in conjunction with Mercurium. Mercurium transforms OmpSs-2 directives in the application code into calls to the Nanos6 API and then invokes the native compiler to process the transformed code.

Furthermore, Nanos6 offers an instrumented version that integrates with ovni. This allows the generation of execution traces, enabling post-mortem analysis with tools like Paraver. The traces provide insights into task and thread execution, revealing the tasks and threads active at any given moment during the execution. Detailed explanations of both tracing tools are presented in the subsequent subsections.

4.1.3 MPICH

MPICH [16], developed at the Argonne National Laboratory and in collaboration with other partners, is a widely used and high-performance implementation of the Message Passing Interface (MPI) standard.

Distributed under an open-source license, MPICH has two primary goals. Firstly, it aims to provide an efficient MPI implementation that supports diverse computation and communication platforms. These include commodity clusters, high-speed networks such as 10 Gigabit Ethernet, InfiniBand, and Myrinet, as well as proprietary high-end computing systems like Blue Gene and Cray. Secondly, MPICH strives to facilitate cutting-edge research in MPI by offering a modular framework that can be extended for the development of derived implementations. In essence, MPICH serves as a high-quality reference implementation of the latest MPI standard, offering reliability and performance.

4.1.4 Ovni

The ovni [17] project introduces an advanced instrumentation library that plays a vital role in capturing and analyzing program execution dynamics. This innovative library efficiently records small events, starting at a mere 12 bytes, as programs run, providing valuable insights into the intricacies of their execution.

The instrumentation process employed by ovni consists of two distinct stages: runtime tracing and emulation. During runtime tracing, the library captures concise binary events that accurately describe the ongoing activities of the program. These events are promptly stored on disk, establishing a comprehensive record of the program's execution flow.

Once the program execution concludes, the recorded events undergo careful reading and processing in the subsequent emulation stage. Leveraging the collected event data, the emulation process meticulously reconstructs the execution sequence, facilitating a faithful reproduction of the program's behavior. Through this emulation process, a final execution trace is generated, encapsulating a detailed account of the program's execution.

An inherent advantage of the ovni project lies in the division between the runtime tracing and emulation stages. This separation enables the performance of resource-intensive computations during the trace generation without causing disruptions to the ongoing runtime execution. The decoupling of these processes allows for efficient analysis and the incorporation of computationally demanding techniques.

Within the ovni library, each event is associated with a specific model, which directly maps to a target library or program. These models serve as independent entities, enabling concurrent instrumentation and analysis. By associating events with models, the ovni project achieves modularity and flexibility, facilitating a focused understanding of program behavior within different program contexts.

Through the utilization of the ovni project's powerful instrumentation library, developers and researchers gain the capability to capture and analyze crucial program events. The runtime tracing, emulation, and model-based approach of ovni empower in-depth investigations into program execution, opening avenues for comprehensive understanding, even in scenarios where direct access to the underlying source code is limited.

4.1.5 Paraver

Paraver, [18] developed at the BSC, is a visualization tool that provides users with a comprehensive view of application behavior through qualitative inspection and enables detailed quantitative analysis of performance issues.

The strength of Paraver lies in two key aspects. Firstly, its trace format is semantic-free, allowing easy integration of new performance data and programming models without requiring modifications to the visualizer. This flexibility enables the tool to adapt to evolving needs.

The second aspect is the programmability of metrics. Paraver offers a wide range of time functions, a filter module, and the ability to combine multiple timelines, empowering users to compute and display numerous metrics based on available data.

In combination with Ovni, Paraver forms a powerful toolset for analyzing application executions. These tools are particularly valuable when applications fall short of expected performance, as they facilitate in-depth investigation of performance issues. Moreover, they provide insights into application behavior, even without prior knowledge of the application's source code.

4.2 Methodology

Due to the time constraints of this project, we opted for a simplified approach rather than applying a more advanced methodology. The chosen approach was the Waterfall Method.

The Waterfall Method, depicted in Figure 4.3, consists of several sequential steps. The first step is gathering requirements, which in our case involved identifying the objectives outlined in Section 1.1. The second step is the design phase, where we planned the architecture and functionality of the TALPF library, as discussed in Chapter 5. The third step is implementation, during which we developed the TALPF library, as described in Chapter 6. The fourth step is testing, which we conducted using the benchmarks presented in Chapter 7. Finally, the fifth and last step is deployment and maintenance, although it falls outside the scope of this project.

While it is recommended to follow these steps linearly, it is important to note that the approach is flexible. If any limitations or challenges arise during implementation, it is possible to revisit the design phase and make necessary adjustments to ensure project success.

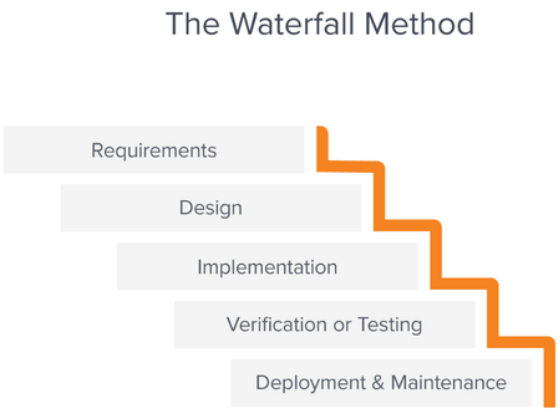


Fig. 4.3: Waterfall Methodology steps.

5 | Task-Aware LPF Design

As we already mentioned, hybrid applications usually have a separation between computation and communication, where the computation phase is working with high internode parallelism using parallel regions or a task-based approach, and the communication is executed with a single-threaded approach. This fork-join approach has several performance losses. First the synchronization point between the computation phase and the communication phase, where some communication may be able to start before all the computation has finished. Second, the serialization of the communication, where, if the library supports it, can be done with a parallel approach, and thus, not wasting resources keeping the other threads idle.

BSP models enforce the separation of computation and communication phases, but they introduce a synchronization phase that incurs overhead due to explicit synchronization using a barrier.

As discussed in Chapter 3, prior research has explored the concept of taskifying and combining the communication and computation phases. These studies have demonstrated the potential to enhance the performance of hybrid applications. Our primary objective is to adapt this approach to LPF, enabling the taskification of both the communication and synchronization phases.

Furthermore, we aim to address the overhead associated with the synchronization phase by introducing new modes that eliminate the need for explicit barriers. These modes will ensure proper synchronization between processes while minimizing unnecessary synchronization overhead.

Figure 5.1 illustrates the software levels involved in a TALPF application. The hybrid application interacts with TALPF, which directly interacts with the *IBVerbs* library for inter-process communication, and the *Nanos6* runtime ensures the proper behavior of communication and synchronization tasks. Unlike interoperability libraries like TAMPI or TAGASPI, TALPF is an adaptation of the LPF library, providing greater flexibility in the design process.

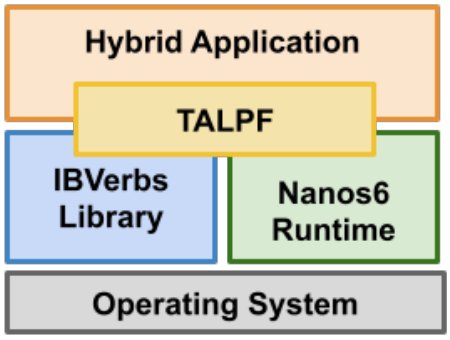


Fig. 5.1: Task-Aware software stack in a hybrid application.

5.1 Task Awareness

To design a Task-Aware library there are some points to take into account. First, all the tasks have to be able to work in parallel. Second, a task should be asynchronous and avoid blocking the task wasting resources waiting for completion that can take a lot of time to reach. Third, if the tasks are asynchronous, there has to be a mechanism that checks if the task is completed.

The first point can be addressed by the use of atomic constructs and critical regions, which only one process can access. The second point primarily affects communication and synchronization, where they issue a communication operation and have to wait for the completion of the operation. Accessing or interacting with remote resources can take a long time, in which the local resources are waiting, usually doing nothing useful, to approach this point, we can make the tasks asynchronous by only issuing the communication operation. Finally, the third point can be addressed with a polling task that is executed periodically to check if the operations have been completed, without waiting for them.

To implement the second and third points, *Nanos6* gives us some mechanisms that allow the tasks to finish, but not release their dependencies, as we will explain in more detail in Chapter 6.

5.1.1 Communication Primitives

Before starting with the Task-Awareness of the communication primitives, our first design choice is to issue directly the communication instead of pushing the operation in a queue to later be executed in the synchronization, as in LPF. Figure 5.3 shows a diagram of the original LPF put (top), and a TALPF put (bottom), where the original LPF put pushes the communication operation into a queue and has a big synchronization, and TALPF put issues directly the operation and the synchronization is smaller. We expect this change to improve the performance, by advancing the beginning of the operation we are also advancing the completion, reducing the time of

the synchronization, as now do not have to issue the operations itself, and part of the time to reach the completion already passed before entering to the synchronization.

This approach has a drawback, as figure 5.3 shows, there is a time between the issue of the communication operation and the synchronization where the content of the remote memory, in the case of a put, or local memory, in the case of a get, is undetermined. In the case of a put, this error can have catastrophic behavior as the local process does not know the state of the other processes and can overwrite data that was still to be processed. To solve this problem the easiest approach is using double buffering.

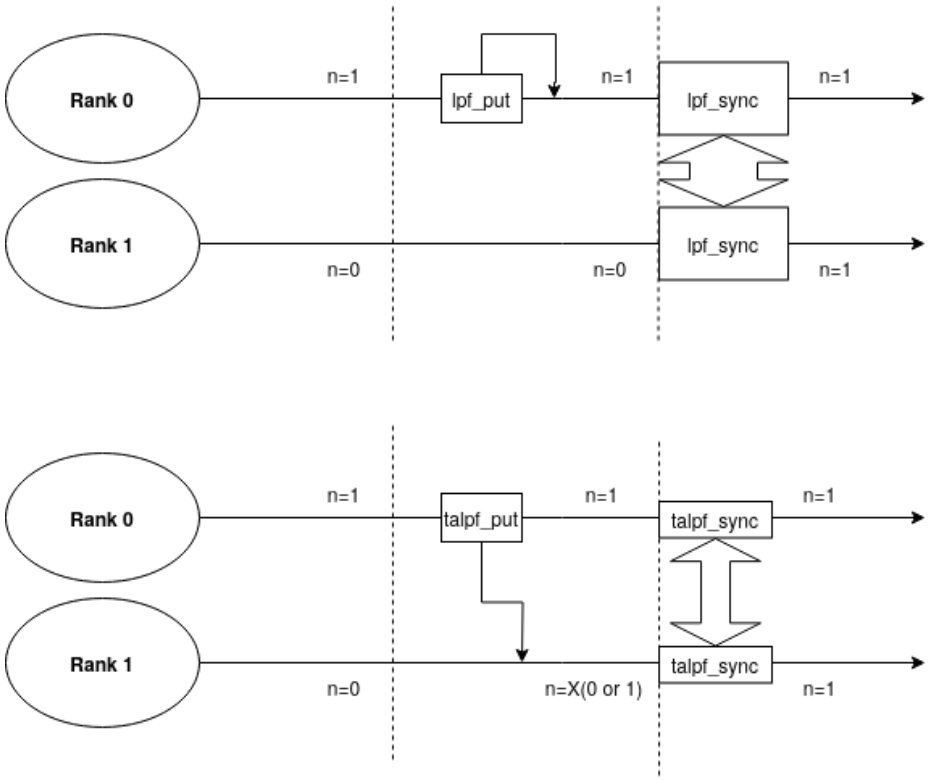


Fig. 5.2: Original LPF put behavior (top) against TALPF put behavior (bottom).

As we expect our tasks to be as asynchronous as possible, our communication primitive cannot wait for the completion of the operation, instead, once the operation is issued, locks the dependencies, and delegates the completion checking to the polling task, that will release the dependences one the operation is completed. This approach allows us to make the communication task asynchronous, but not allow the dependent tasks to be executed before the operation is completed.

Finally, to avoid any confusion with the original LPF primitives, we changed the primitives' name from `lpf_put` and `lpf_get` to `talpf_put` and `talpf_get`.

```
lpf_err_t talpf_put(
       lpf_t ctx, lpf_memslot_t src_slot,
       size_t src_offset, lpf_pid_t dst_pid,
       lpf_memslot_t dst_slot, size_t dst_offset,
       size_t size, lpf_msg_attr_t attr
);
lpf_err_t talpf_get(
       lpf_t ctx, lpf_pid_t src_pid,
       lpf_memslot_t src_slot, size_t src_offset,
       lpf_memslot_t dst_slot, size_t dst_offset,
       size_t size, lpf_msg_attr_t attr
);
```

5.1.2 Synchronization Primitive

The design of the synchronization primitive focus mainly on simplification and asynchronicity. Like the communication primitives, we want to avoid having a task doing nothing while waiting for the communication to complete, with similar approaches.

The original LPF consists of several phases, a first meta-data exchange, a write-conflict resolution with a second meta-data exchange, the data exchange, waiting for the local completions, and a final barrier.

In TALPF, we issue the communication operations in the communication primitives, therefore the write-conflict resolution is not used, as the operations already started, and if there was a conflict it is too late to resolve it. However, in scenarios where multiple processes need to write to the same memory, users can utilize the atomic primitives introduced in the previous section to handle these situations.

The synchronization task should be dependent on all the communication tasks, therefore all the local communication operations should have been completed by the time the synchronization primitive is executed. And thus we do not need to wait for local completions.

As we will explain in Section 5.3, we designed new modes without a barrier, so the new synchronization needs to wait only for the remote completions.

The results of these changes give a synchronization that performs a metadata exchange, exchanges the number of messages between processes, waits for remote completion, and executes a barrier. As there are many phases, it is difficult to make them all asynchronous, but like the communication primitives, we can make use of the polling task to check for completions, and avoid wasting resources while waiting to complete the operations.

5.1.3 Polling Task

The Polling Task is a task that is executed periodically, as the communication and synchronization tasks, we want to avoid any type of blocking operation, making the polling task fast and less resource-consuming. It is a key element in TALPF, as it handles all the asynchronicity of the communication and synchronization tasks.

The Polling task has three main functions: checking for local completions and releasing its dependences once completed; checking for remote completions; and handle the asynchronicity of the synchronization.

5.2 Atomic Communication

We designed new atomic communication primitives, to handle cases where two or more ranks have to write to the same memory, for example in a reduction where you want a process to have the addition of the values of all the processes.

The design is similar to the other communication primitives, but by limitation of the implementation, in this case, instead of writing/reading a variable section of the memory region, we focus only on integers of 64 bits.

The two atomic primitives added are: atomic fetch and add, `talpf_atomic_fetch_and_add`, where the user has to pass a value that will be added to the destination memory slot; and atomic compare and swap, `talpf_atomic_cmp_and_swp`, where the user has to pass the value to compare `cmp` and the value that will be swapped in case the `cpm` value and the value in the destination memory are the same.

```
lpf_err_t talpf_atomic_fetch_and_add(
       lpf_t ctx, lpf_memslot_t src_slot,
       size_t src_offset, lpf_pid_t dst_pid,
       lpf_memslot_t dst_slot, size_t dst_offset,
       uint64_t value
);
lpf_err_t talpf_atomic_cmp_and_swp(
       lpf_t ctx, lpf_memslot_t src_slot,
       size_t src_offset, lpf_pid_t dst_pid,
       lpf_memslot_t dst_slot, size_t dst_offset,
       uint64_t cmp, uint64_t swp
);
```

5.3 New Synchronization Modes

Synchronizing all the processes is a very costly operation, for this reason, one of the main goals in optimizing the performance of TALPF is simplifying the synchronization, but this approach is not enough, so we designed new synchronization modes that minimize the overhead of the synchronization by removing the all-to-all communication between processes, such as the metadata exchange, the number of messages exchanged, and the barrier.

We named these new modes Zero-Cost Synchronization, as they do not add any extra communication between processes, and only do the strictly necessary to synchronize with the processes that are directly communicating. As we will explain these new methods rely on the repetitive behavior of the application, such as an iterative application, or redirect all the responsibility to the user, reducing the programmability.

5.3.1 Cached Mode

The first of these new modes is the *cached* mode, where the first iteration behaves similarly to the *default* mode and saves the number of messages that has to wait for future iterations, in the following iterations, instead of exchanging the number of messages, it takes the value of the first iteration. This method avoids the metadata exchange and the final global barrier to truly be a Zero-Cost Synchronization. Note that this mode only works in a repetitive environment, where all the synchronizations have to be identical in the number of messages exchanged.

5.3.2 Msg Mode

The second of these new modes is the *msg* mode, where the user explicitly indicates the number of messages that the process will receive. This mode avoids the metadata exchange, all the exchanges of the number of messages, and the final barrier to be a Zero-Cost Synchronization mode, but adds more complexity to the application programmability. This mode can work in a non-repetitive environment, but it does not have good programmability compared to the *default* and *cached* modes.

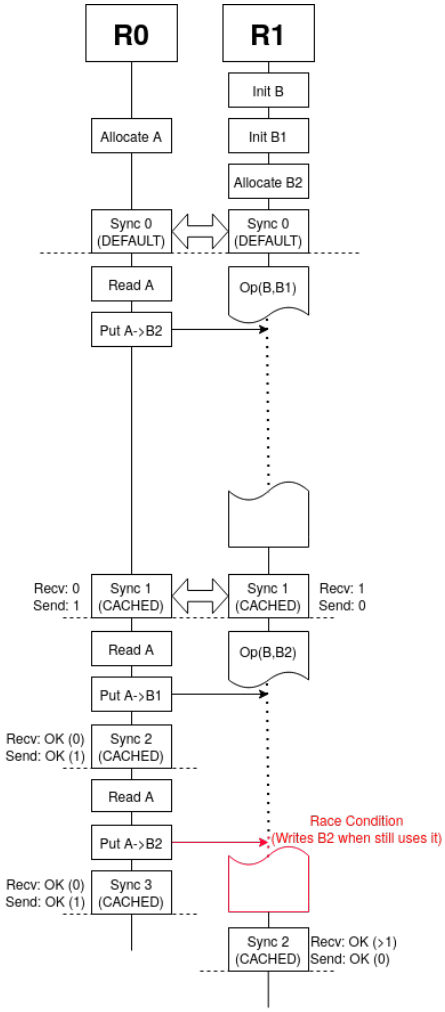
5.3.3 Problems

The Zero-Cost Synchronization only works when the communication between ranks is bidirectional. If there is no bidirectional communication between ranks, data-races might occur, as depicted in Figure 5.3. This problem occurs because the process is not waiting for any message from the receiving process, so does not know anything about its state, this problem can be solved by adding an empty message to synchronize the two ranks, the cost of this message is negligible compared to a global synchronization, but decreases the programmability of the Zero-Cost Synchronization modes.

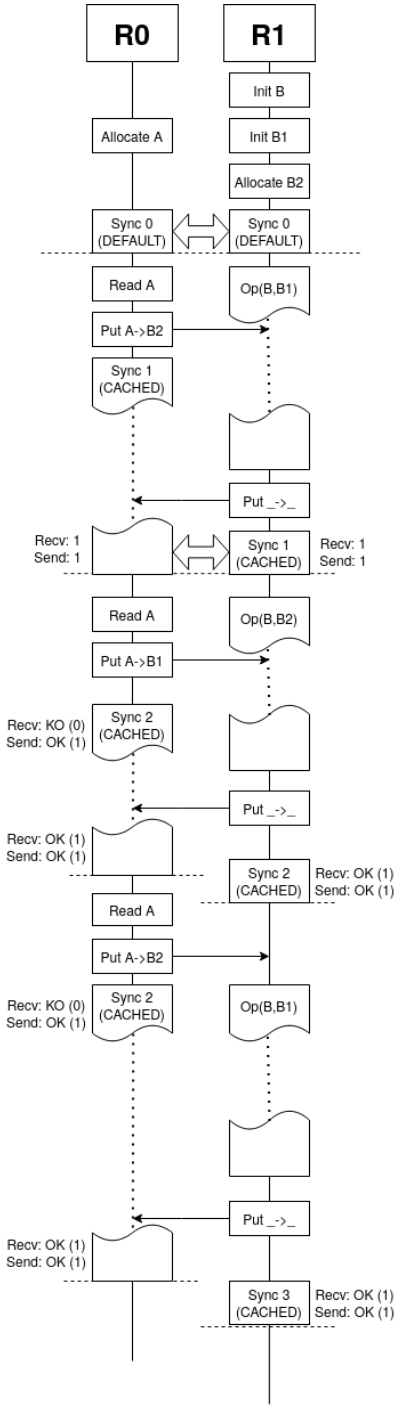
Note that this is not a problem if the application already has a bidirectional communication pattern.

5.3.4 Barrier Modifier

Another way to solve the Zero-Cost Synchronization modes problem is to add a global barrier at the end of the synchronization, of course, then it is not a Zero-Cost Synchronization mode, but it has some performance benefits, as it avoids the metadata exchange and the number of messages exchange.



(a) Cached mode without bidirectional communication.



(b) Cached mode with bidirectional communication.

Fig. 5.3: Application diagrams showing that the Zero-Cost synchronization without bidirectional communication (a) can reach a race condition; while with bidirectional communication (b) avoids the data race and maintains the synchronization flexibility

6 | Task-Aware LPF Implementation

In this chapter, we will introduce the implementation details of the design of Chapter 5. As all the communication primitives share a similar structure, we will introduce the implementation of all the communication primitives together and explain the main differences. Similarly, we will explain the synchronization primitive and all its modes in one section, as they share most of its implementation. Finally, we will explain further the polling task and all its functionalities.

In Figure 6.1 we can observe a graphic of the interaction between TALPF and *Nanos6*, where the TALPF blocks represent the polling task, and the rest a normal execution of *Nanos6*, where the tasks can have mainly four states. The ready state is where the task is in the ready queue waiting for the scheduler to assign a CPU. The running state is where the task is executed in a CPU. The finished state is where the task already finished its execution, but still, it is not completed, as still the dependencies are blocked. The block state is where the task is waiting for its dependencies to be released. The polling task is used to hold the dependencies until the operations are completed, in the case of the communication operations, waiting for the completion, and in the synchronization operation to finish the synchronization itself.

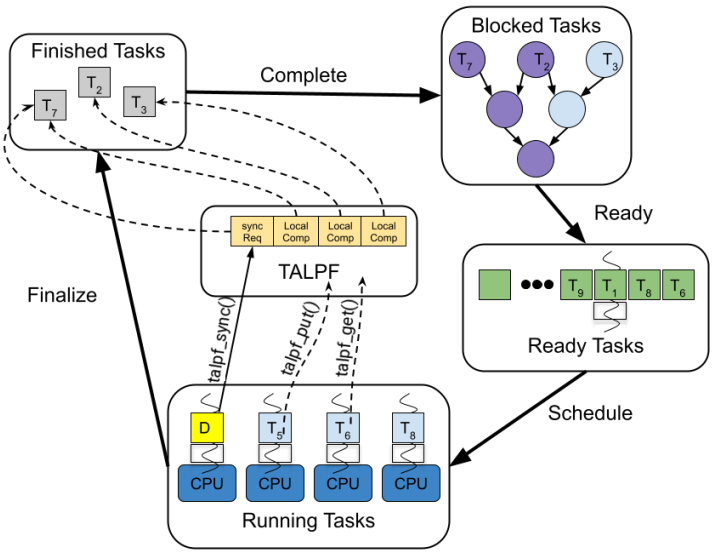


Fig. 6.1: Graph of the *Nanos6* runtime combined with TALPF.

6.1 Communication Primitives

The implementation of the communication primitives consists primarily in posting the communication operation directly, instead of pushing the operation to an internal queue. With this change, the asynchronicity is implicit, in the communication primitive we only post the operation, and the completion checking is done in the polling task. All the communication primitives share a similar structure, the setup, the message preparation, and the post of the operation.

In the setup, we define the basic structures, such as the Send Work Requests and the SGE object, for *IBVerbs*. We also increase a counter of the total local number of communication operations and a counter of local communication operations to a specific process. After that, we increase the *Nanos6* counter.

In `talpf_put` and `talpf_get` the operation might need a variable number of Send Work Requests depending on the memory size of the operation. In `talpf_get`, `talpf_atomic_fetch_and_add` and `talpf_atomic_cmp_and_swp` the operation needs an extra Send Work Request to put the immediate value, as *IBVerbs* only supports immediate values in the writes.

```
1  const MemorySlot & src = m_memreg.lookup( srcSlot );
2  const MemorySlot & dst = m_memreg.lookup( dstSlot );
3
4  ASSERT( dst.mr );
5
6  int numMsgs = size/m_maxMsgSize + (size % m_maxMsgSize > 0); //+1 if last
       msg size < m_maxMsgSize
7
8  struct ibv_sge sges[numMsgs+1]; //+1 for the extra message
9  struct ibv_send_wr srs[numMsgs+1]; //+1 for the extra message
10
11 struct ibv_sge *sge;
12 struct ibv_send_wr *sr;
13
14 atomic_fetch_add(&m_numMsgs, 1);
15 atomic_fetch_add(&m_numMsgsSync[srcPid], 1);
16 void *counter = nanos6_get_current_event_counter();
17 nanos6_increase_current_task_event_counter(counter, 1);
```

Code 6.1: Setup of the `talpf_get` implementation.

In the message preparation, we initialize the Send Work Request and its SGEs. The SGE contains the address, length, and key of the local memory region key. The Send Work Request contains a pointer to the next Send Work Request if there is any, the send flags, the Work Request ID set to the pointer of the *Nanos6* counter, a pointer to the SGE, the Operation Code, the immediate value, the remote address, and the remote memory region key.

If the message in the `talpf_put` and the `talpf_get` primitives is too long it will be split into different Send Work Requests. Only the last message produces a local and remote completion. To achieve that, we set the send flags to `IBV_SEND_SIGNALED` and the operation code to `IBV_WR_RDMA_WRITE_WITH_IMM`. Note that the operation flag is a write, so for the operations, except the `talpf_put`, we need to add an extra Send Work Request with an empty write.

```
for(int i = 0; i< numMsgs; i++){
      sge = &sges[i]; std::memset(sge, 0, sizeof(ibv_sge));
      sr = &srs[i]; std::memset(sr, 0, sizeof(ibv_send_wr));
      const char * localAddr
            = static_cast<const char *>(dst.glob[m_pid].addr) + dstOffset;
      const char * remoteAddr
            = static_cast<const char *>(src.glob[srcPid].addr) + srcOffset;

      sge->addr = reinterpret_cast<uintptr_t>( localAddr );
      sge->length = std::min<size_t>(size, m_maxMsgSize );
      sge->lkey = dst.mr->lkey;

      sr->next = &srs[i+1];
      sr->send_flags = 0;
      sr->wr_id = m_pid;
      sr->sg_list = sge;
      sr->num_sge = 1;
      sr->opcode = IBV_WR_RDMA_READ;
      sr->wr.rdma.remote_addr = reinterpret_cast<uintptr_t>( remoteAddr );
      sr->wr.rdma.rkey = src.glob[srcPid].rkey;

      size -= sge->length;
      srcOffset += sge->length;
      dstOffset += sge->length;
}

// add extra "message" to do the local and remote completion
sge = &sges[numMsgs]; std::memset(sge, 0, sizeof(ibv_sge));
sr = &srs[numMsgs]; std::memset(sr, 0, sizeof(ibv_send_wr));

const char * localAddr = static_cast<const char *>(dst.glob[m_pid].addr);
const char * remoteAddr = static_cast<const char *>(src.glob[srcPid].addr);

sge->addr = reinterpret_cast<uintptr_t>( localAddr );
sge->length = 0;
sge->lkey = dst.mr->lkey;

sr->wr_id = (uint64_t)(counter);
sr->next = NULL;
sr->send_flags = IBV_SEND_SIGNALED;
sr->sg_list = sge;
sr->num_sge = 0;
sr->opcode = IBV_WR_RDMA_WRITE_WITH_IMM; // There is no READ_WITH_IMM
sr->imm_data = m_sync_counter;
sr->wr.rdma.remote_addr = reinterpret_cast<uintptr_t>( remoteAddr );
sr->wr.rdma.rkey = src.glob[srcPid].rkey;
```

Code 6.2: Message Preparation of the `talpf_get`

Finally, to post the operation we call `ibv_post_send` with the Queue Pair related to the destination and the first Send Work Request. if it fails, we decrease the local number of message counters and we throw an exception.

```
struct ibv_send_wr *bad_wr = NULL;
int err;
if (err = ibv_post_send(m_connectedQps[srcPid].get(), &srs[0], &bad_wr))
{
      atomic_fetch_sub(&m_numMsgs, 1);
      atomic_fetch_sub(&m_numMsgsSync[srcPid], 1);

      LOG(1, "Error while posting RDMA requests: " << std::strerror(err) );
      throw Exception("Error while posting RDMA requests");
}
```

Code 6.3: Post of the opperation of the `talpf_get` implementation.

6.2 Synchronization Primitive

The implementation of the synchronization primitive consists primarily in making most of the primitive asynchronous and simplifying the heavy operations such as the metadata exchange. It also includes all the modes defined in Section 5.3, which only affects the number of messages exchange implementation. In TALPF the synchronization primitive does not execute the synchronization itself, only the exchanges, instead creates a `SyncRequest` struct with all the information necessary for the polling task to execute the synchronization asynchronously. The primitive consists of three parts, the meta-data exchange, the number of messages exchanged, and the `SyncRequest` preparation.

The metadata exchange is only executed with the *default* mode, and basically, the only function it does is exchanging two values to know if some process needs to reconnect or has aborted. We avoid this metadata exchange in the *cached* and *msg* modes for performance, and because we expect these modes to be used only in the compute part of the application, for the setup and initialization we expect only the *default* mode.

The metadata exchange is done through a `MPI_Iallreduce`, this call is asynchronous, to avoid wasting time waiting for the completion, we block the task with the *Nanos6* block API, and let the polling task unblock the synchronization task once the `MPI_Iallreduce` is completed.

```
1
2  int sync_mode = attr & LPF_SYNC_MODE;
3  int sync_value = attr & ~(LPF_SYNC_MODE | LPF_SYNC_BARRIER);
4  int sync_barrier = ((attr & LPF_SYNC_BARRIER) != 0) | (sync_mode ==
       LPF_SYNC_DEFAULT);
5
6  int voted[2];
7
8  if(sync_mode == LPF_SYNC_DEFAULT){
9        voted[0] = 0;
10       voted[1] = 0;
11
12       m_comm.getRequest(&m_blockRequest);
13       m_comm.iallreduceSum(vote, voted, 2, m_blockRequest);
14       m_blockContext = nanos6_get_current_blocking_context();
15       nanos6_block_current_task(m_blockContext);
16
17       if (voted[0] != 0 ) {
18             vote[0] = voted[0];
19             return;
20       }
21
22       if (voted[1] > 0){
23             reconnectQPs();
24       }
25 }
```

Code 6.4: Metadata exchange of the `talpf_sync` implementation.

The number of messages exchanged has different behaviors depending on the mode, where the only exchange happens in the *default* mode. The *cached* mode in the first call executes the *default* mode exchange and stores the value, then, in the following calls reuses the same value without making any new exchange. The *default* mode exchanges the number of messages with a blocking `MPI_Allreduce` for each process, as the processes are already synchronized from the metadata exchange this collective does not have a performance loss. Once the exchange is completed, if it was not in *cached* mode, invalidates the cached value. For the *msg* mode, we simply take the value from the parameter.

```
1  int remoteMsgs = 0;
2  int numMsgsSync[m_nprocs];
3
4  for(int i = 0; i < m_nprocs; i++) {
5        numMsgsSync[i] = m_numMsgsSync[i];
6        m_numMsgsSync[i] = 0;
7  }
8
9  switch (sync_mode){
```

```
10        case LPF_SYNC_CACHED:
11              if (m_sync_cached){
12                    remoteMsgs = m_sync_cached_value;
13                    break;
14              }
15              //else
16              m_sync_cached = true;
17              // fall through
18        case LPF_SYNC_DEFAULT:
19              //get num remote completions
20              for(int i = 0; i < m_nprocs; i++){
21                    if(i == m_pid) remoteMsgs = m_comm.allreduceSum(numMsgsSync[
                          i]);
22                    else m_comm.allreduceSum(numMsgsSync[i]);
23              }
24
25              if(sync_mode == LPF_SYNC_DEFAULT) m_sync_cached = false;
26              m_sync_cached_value = remoteMsgs;
27              break;
28
29        case LPF_SYNC_MSG(0):
30              remoteMsgs = sync_value;
31              break;
32  }
```

Code 6.5: Number of messages exchange of the `talpf_sync` implementation.

Finally, `SyncRequest` preparation checks if there is an active `syncRequest`, which by design and if the dependences are correct should be impossible, if there is one it aborts the program. After that checking, it set the `syncRequest` with the different values. If the synchronization primitive is in *default* mode or has the barrier flag, the `syncRequest.withBarrier` parameter is set to true. The `syncRequest.remoteMsgs` is set to the number of messages that we previously defined. And the `syncRequest.counter` contains the *Nanos6* counter, which is previously initialized. Finally, we activate the `syncRequest` with `syncRequest.isActive` equal to true, which will allow the polling task to start processing the `syncRequest`.

```
if(syncRequest.isActive) {
      LOG(1, "There is another sync request active");
      throw Exception("There is another sync request active");
}

syncRequest.withBarrier = sync_barrier;
syncRequest.remoteMsgs = remoteMsgs;

void *counter = nanos6_get_current_event_counter();
nanos6_increase_current_task_event_counter(counter, 1);
syncRequest.counter = counter;

syncRequest.isActive = true;
```

Code 6.6: Preparation of the `SyncRequest` object of the `talpf_sync` implementation.

6.3 Polling Task

Nanos6 does not offer any API with a polling task, instead offers an API to create tasks and a sleep function, with these two we can create a task with a loop that will be only ended at the end of the execution, this loop is executed once every `pollingFrequency`, simulating a real polling task. The polling frequency value can be modified via an environment variable called `TALPF_POLLING_FREQUENCY`. To terminate the polling task, while destroying the TALPF objects the `stopPollingTask` function is called, which changes the condition of the polling task's while to be false, therefore in the next iteration the polling task will terminate. From this loop, the polling task calls the different functionalities, the local competition polling, the remote completion polling, the block handler, and the synchronization handler.

,

```
void pollingTask(){
      while(!m_stopPollingTask){
            doLocalProgress();
            doRemoteProgress();
            if(m_blockContext != NULL)
                  processBlock();
            if(syncRequest.isActive)
                  processSyncRequest();
            nanos6_wait_for(pollingFrequency);
      }
}

void stopPollingTask(){
      m_stopPollingTask = 1;
}
```

Code 6.7: TALPF polling task main loop and termination function.

The local completion polling handles all the local completions generated by the communication primitives, it polls the completion queue using the `ibv_poll_cq` call form *IBVerbs*, in batches of `POLL_BATCH`. If there are fewer completions than `POLL_BATCH` the polling task assumes there are no mere completions so it stops polling for this iteration, if it reaches a maximum number of completions, defined by `MAX_POLLING`, the polling task also stops polling for local completions as we want to do progress in the other functionalities and not stall in one. Both `POLL_BATCH` and `MAX_POLLING` is defined in the same file, by default their values are 8 and 128 respectively.

With the completion queue already polled, the completions are processed. In case of an error, the polling task throws an exception. If the completion is correct, we retrieve the *Nanos6* counter from the Work Request ID, which we previously set to the pointer of the *Nanos6* counter in the communication primitive, and we decrease it, releasing the dependencies of the communication task.

```
void IBVerbs :: doLocalProgress(){

      int n = m_numMsgs;
      int error = 0;
      struct ibv_wc wcs[POLL_BATCH];

      int pollResult, totalResults = 0;

      LOG(5, "Polling for " << n << " messages" );

      do {
            pollResult = ibv_poll_cq(m_cqLocal.get(), POLL_BATCH, wcs);
            if (pollResult > 0) {
                  totalResults += pollResult;
                  LOG(4, "Received " << pollResult << " acknowledgements");
```

```
                atomic_fetch_sub(&m_numMsgs, pollResult);

                for (int i = 0; i < pollResult ; ++i) {
                     if (wcs[i].status != IBV_WC_SUCCESS)
                     {
                           LOG( 2, "Got bad completion status from IB
                               message."
                            " status = 0x" << std::hex << wcs[i].status
                            << ", vendor syndrome = 0x" << std::hex
                            << wcs[i].vendor_err );
                          error = 1;
                     }
                     else {
                          nanos6_decrease_task_event_counter((void *)wcs[i
                              ].wr_id, 1);
                     }
                }
           }
           else if (pollResult < 0)

           {
                LOG( 1, "Failed to poll IB completion queue" );
                throw Exception("Poll CQ failure");
           }
           if (error) {
                throw Exception("Error occurred during polling");
           }
     } while (pollResult == POLL_BATCH && totalResults < MAX_POLLING);
}
```

Code 6.8: TALPF polling task local completion polling.

The remote completion polling handles all the remote completions, which are generated when a remote communication, or local if the process communicates to itself, is completed. These completions consume Receive Work Requests, which are refilled once the completion is processed to be able to receive communication operations without any errors. Similarly to the local completion polling, we poll the completion in batches of `POLL_BATCH` with a maximum of `MAX_POLLING` completions per polling task iteration.

The completion is processed by taking the immediate data of the Work Request, which previously was initialized in the corresponding communication operation with the synchronization ID, incrementing the corresponding recv count, to later use it in the sync, and posting a new Receive Work Request to the Shared Receive Queue.

```
void IBVerbs :: doRemoteProgress(){

     struct ibv_wc wcs[POLL_BATCH];

     struct ibv_recv_wr wr;
```

```
6       struct ibv_sge sg;
7       struct ibv_recv_wr *bad_wr;
8
9       sg.addr = (uint64_t) NULL;
10      sg.length = 0;
11      sg.lkey = 0;
12      wr.next = NULL;
13      wr.sg_list = &sg;
14      wr.num_sge = 0;
15      wr.wr_id = 0;
16
17      int pollResult, totalResults = 0;
18
19      do {
20            pollResult = ibv_poll_cq(m_cqRemote.get(), POLL_BATCH, wcs);
21            for(int i = 0; i < pollResult; i++){
22                  m_recvCounts[wcs[i].imm_data%1024]++;
23                  ibv_post_srq_recv(m_srq.get(), &wr, &bad_wr);
24            }
25            if(pollResult > 0) totalResults += pollResult;
26      } while (pollResult == POLL_BATCH && totalResults < MAX_POLLING);
27  }
```

Code 6.9: TALPF polling task remote completion polling.

The synchronization primitive relies upon the polling task with two things. The first handles the asynchronicity of the metadata exchange and waits for the completions.

The metadata exchange asynchronicity, as we have seen in the previous section, it is handled by blocking the task from *Nanos6*. The polling task tests the completion in each iteration, and if the test returns that the operation is completed, we unlock the synchronization task using the `nanos6_unblock_task` primitive with the task context.

```
1   void IBVerbs :: processBlock(){
2       int flag;
3       m_comm.test(m_blockRequest, &flag);
4
5       if(flag == 1){
6             free(m_blockRequest);
7             m_blockRequest = NULL;
8
9             void *context = m_blockContext;
10            m_blockContext = NULL;
11            nanos6_unblock_task(context);
12      }
13  }
```

Code 6.10: TALPF polling task metadata exchange block handler.

Finally, the synchronization handler is divided into two phases, the wait phase and the barrier phase. The first phase, the wait phase, checks if the number of received messages is at least the expected number. Only once the condition is true, resets the value of the remote messages counter and expected messages. In the modes without a barrier, it completes the synchronization directly, and in the modes with a barrier initiates an asynchronous barrier. The second phase, the barrier phase, is only accessed if the mode has a barrier, and it tests the completion of the barrier. Once the barrier is completed, complete the synchronization. To complete the synchronization, it deactivates the `syncRequest` and decreases the *Nanos6* counter of the synchronization task.

```
1  void IBVerbs :: processSyncRequest(){
2        int flag;
3        if(syncRequest.withBarrier && syncRequest.secondPhase) {
4              m_comm.test(syncRequest.barrierRequest, &flag);
5              if(flag == 1){
6                    void *counter = syncRequest.counter;
7                    syncRequest.counter = NULL;
8                    syncRequest.secondPhase = false;
9                    syncRequest.isActive = false;
10                   m_sync_counter++;
11                   nanos6_decrease_task_event_counter(counter, 1);
12             }
13       }
14
15       // wait for remote completions
16       else if (m_recvCounts[m_sync_counter%1024] >= syncRequest.remoteMsgs){
17             if (m_recvCounts[m_sync_counter%1024] > syncRequest.remoteMsgs)
                   //Print warning
18                   LOG(1, "There are more remote completions than the expected"
                         );
19             m_recvCounts[m_sync_counter%1024] -= syncRequest.remoteMsgs;
20             syncRequest.remoteMsgs = 0;
21             if (syncRequest.withBarrier) {
22                   m_comm.ibarrier(syncRequest.barrierRequest);
23                   syncRequest.secondPhase = true;
24             }
25
26             else {
27                   void *counter = syncRequest.counter;
28                   syncRequest.counter = NULL;
29                   syncRequest.isActive = false;
30                   m_sync_counter++;
31                   nanos6_decrease_task_event_counter(counter, 1);
32             }
33       }
34 }
```

Code 6.11: TALPF polling task synchronization handler.

7 | Evaluation

In this chapter, we will test our implementation against TAMPI using four benchmarks, a Jacobi 2D heat diffusion, a Gauss-Seidel 2D heat diffusion, an N-body particle simulation, and a Matrix Multiplication executed several times. These implementations are meant only to make the comparison between TAMPI and TALPF, they are not meant to achieve the peak performance on either, as we wanted to compare them in equal conditions, so we could not provide a specific optimization that only worked on one of them.

We also wanted to implement a naive solution, simulating a normal use of the library. For that reason, we used to power two sizes in the problems, even if we know that the CPU topology is not a power of two and therefore affects the peak performance.

The only benchmark where we tweaked some *Nanos6* parameters and we did not use a naive implementation (we added priorities to tasks to overcome the topology problem) is in the N-body simulation, where the naive implementation was terrible.

7.1 Environment

We leverage up to four compute nodes of the Huawei cluster at BSC to run the experimental evaluation. Each compute node is equipped with 2 sockets of Kunpeng 920 processors, which are based on the ARMv8.1 architecture, with 64 cores each, totaling 128 cores per node, and 256 GB of memory. We use a configuration of four processes per node because each node features four NUMA domains, two per socket. Each process runs on its own NUMA domain and has 32 cores available. Due to system noise detected in the machine, to minimize this system noise, we decided to use 31 cores instead.

In both TAMPI and TALPF the polling service is set to the same value of 100 ns, and, except for the N-body simulation benchmark, all the benchmarks are executed with the default *Nanos6* configuration. In the N-body simulation, we modified the immediate successor probability to zero.

We developed several TALPF versions to test the new synchronization. All the versions have the computation, communication, and synchronization phases taskified, where the main change between versions is the synchronization mode. Each benchmark has its implementation of the version. The versions we have developed (except *TAMPI*) are:

- **TAMPI:** The communication phase uses tasks calling the non-blocking and asynchronous communication services of TAMPI for exchanging messages with the neighbor ranks. The tasks call `MPI_Isend` and `MPI_Irecv` for issuing the communications, and then, bind the task completion to the generated MPI requests through the `TAMPI_Iwait` function.

- **TALPF (DEFAULT):** The communication phase instantiates tasks that execute asynchronous *puts* (`talpf_put`) for exchanging messages. Each iteration instantiates a synchronization task that calls the `talpf_sync` primitive with the `TALPF_SYNC_DEFAULT` mode. For simplicity, we will refer to it as **TALPF(D)**.

- **TALPF (CACHED):** The communication phase instantiates tasks that execute asynchronous *puts* (`talpf_put`) for exchanging messages. Each iteration instantiates a synchronization task that calls the `talpf_sync` primitive with the `TALPF_SYNC_CACHED` mode. For simplicity, we will refer to it as **TALPF(C)**.

- **TALPF (CACHED|BARRIER):** The communication phase instantiates tasks that execute asynchronous *puts* (`talpf_put`) for exchanging messages. Each iteration instantiates a synchronization task that calls the `talpf_sync` primitive with the `TALPF_SYNC_CACHED` mode, with the `TALPF_SYNC_BARRIER` modifier. For simplicity, we will refer to it as **TALPF(C|B)**.

- **TALPF (CACHED with ACK):** The communication phase instantiates tasks that execute asynchronous *puts* (`talpf_put`) for exchanging messages. Each iteration instantiates a synchronization task that calls the `talpf_sync` primitive with the `TALPF_SYNC_CACHED` mode, and adding manually a message to make the communication bidirectional. For simplicity, we will refer as to it **TALPF(C w/ ACK)**.

- **TALPF GET (DEFAULT):** The communication phase instantiates tasks that execute asynchronous *gets* (`talpf_get`) for exchanging messages. Each iteration instantiates a synchronization task that calls the `talpf_sync` primitive with the `TALPF_SYNC_DEFAULT` mode. For simplicity, we will refer to it as **TALPF GET(D)**.

All versions follow the same pattern, although TAMPI features fine-grained synchronizations and TALPF requires the synchronization task. Note that we did not use the `TALPF_SYNC_MSG` mode, as our benchmarks have a clear iterative pattern, in these cases using the `TALPF_SYNC_MSG` mode adds an unnecessary programmability overhead.

Table 7.1 shows the versions used for each benchmark. Note that only Jacobi can use the cached version alone, as it is the only benchmark with bidirectional communication by default, the other benchmarks (Nbody and Matrix Multiplication) have a ring pattern communication, where the communication only goes in one direction.

	Jacobi	Gauss-Seidel	Nbody	MatMul
TAMPI	X	X	X	X
TALPF(D)	X	X	X	X
TALPF(C)	X			
TALPF(C|B)			X	X
TALPF(C w/ ACK)			X	X
TALPF GET(D)			X(*)	

Table 7.1: Table of the versions used in each benchmark.
* Only with the priority version.

7.2 Heat-Diffusion: Jacobi

The heat-diffusion problem [19] consists of a region, usually a matrix, where there are several heat points. The goal of the problem is to diffuse this heat until it reaches a stable point. The heat diffusion solvers, such as Jacobi or Gauss-Seidel, are usually iterative, where in each iteration, the solver diffuses the heat to adjacent cells of the matrix. If it reaches a stable configuration, it halts the execution.

The Jacobi solver approach consists of two matrices, where one matrix contains the current state, and the second matrix is used to store the result of the heat diffusion operation, in the next iteration the role of each matrix is swapped. With this approach, there are no dependencies between computation tasks in the same iteration, as the source is never modified, and the destinations are different for each task.

To use multiple processes, the usual approach is to divide the matrix, where each rank gets a contiguous space of the matrix. The communication pattern in this approach is to exchange the halos to the neighbor processes, as both sides of the halos are exchanged in every iteration, the communication is bidirectional.

Figure 7.1 shows the strong scalability test, from 1 rank to 32 ranks. Both versions of TALPF outperform TAMPI with fewer ranks, but the DEFAULT version of TALPF does not scale well over 4 ranks, the CACHED version keeps a good performance until 32 ranks. In general, all three versions have a good strong scalability only losing in the worst case around a 15% of the performance.

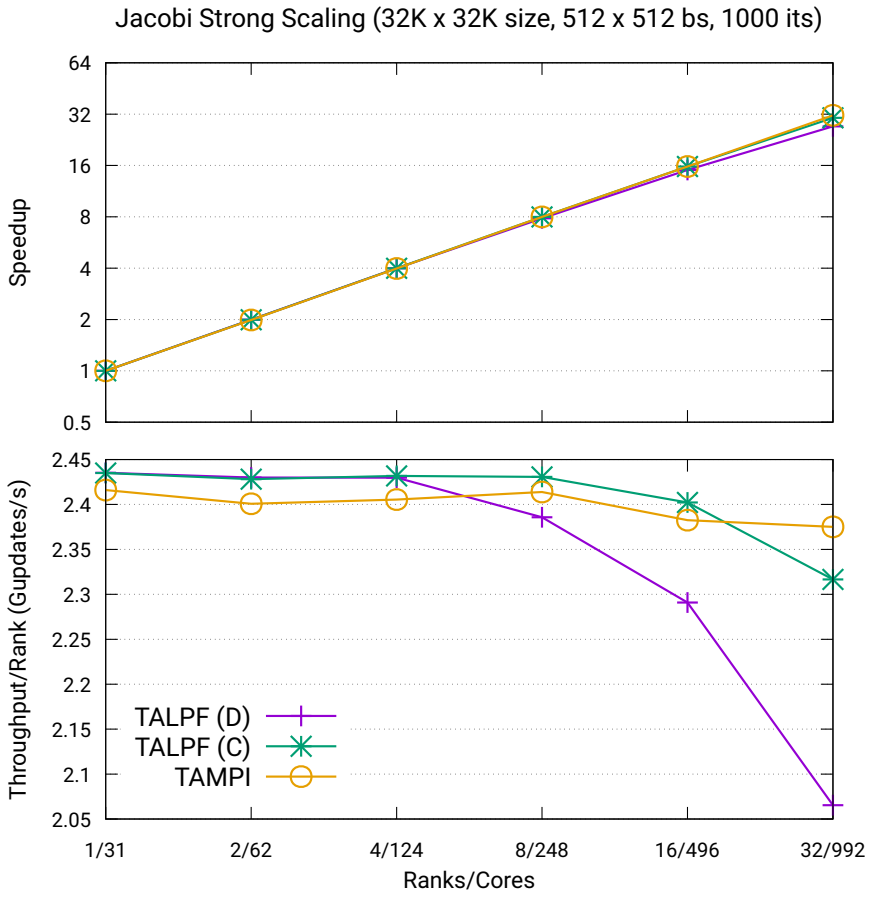


Fig. 7.1: Jacobi Strong Scalability test, showing the speedup (top) and the normalized throughput (bottom). The Purple line is the TALPF(D) version, the green line is the TALPF(C) version, and the yellow line is the TAMPI version.

Figure 7.2 shows an iteration of 4 ranks of 32 ranks total, with the same parameters as the Strong Scalability test. As the figure shows there are four ”batches” of tasks, then there is a fifth with only 4 tasks, and finally a separation between the iterations which correspond to the synchronization. As the size of the problem is a power of two and the number of cores is 31 it cannot fit the four tasks with the first four tasks ”batches” causing a loss of performance. TAMPI can overcome its load imbalance by using a fine-grain synchronization, that allows the application to start the next iteration before finishing the current one.

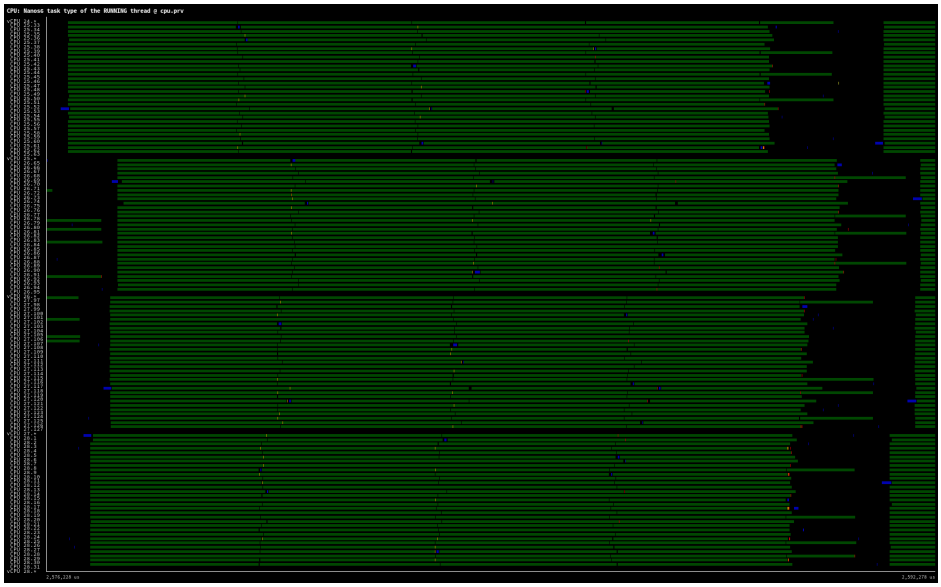


Fig. 7.2: Jacobi trace of the TALPF(C) version with 32 ranks. The image corresponds to an iteration of 4 middle ranks. The green color corresponds to compute tasks, the blue to the polling task, the red and yellow to the communication tasks. The synchronization task is negligible and cannot be seen in this level of zoom.

Figure 7.3 shows the Weak Scalability test, where in the Strong Scalability test the TALPF versions lost performance with a large number of processes, in the Weak Scalability all the versions keep good scalability. Through some investigation, we deduced that this improvement in the Strong Scalability is due to the problem shown in Figure 7.2.

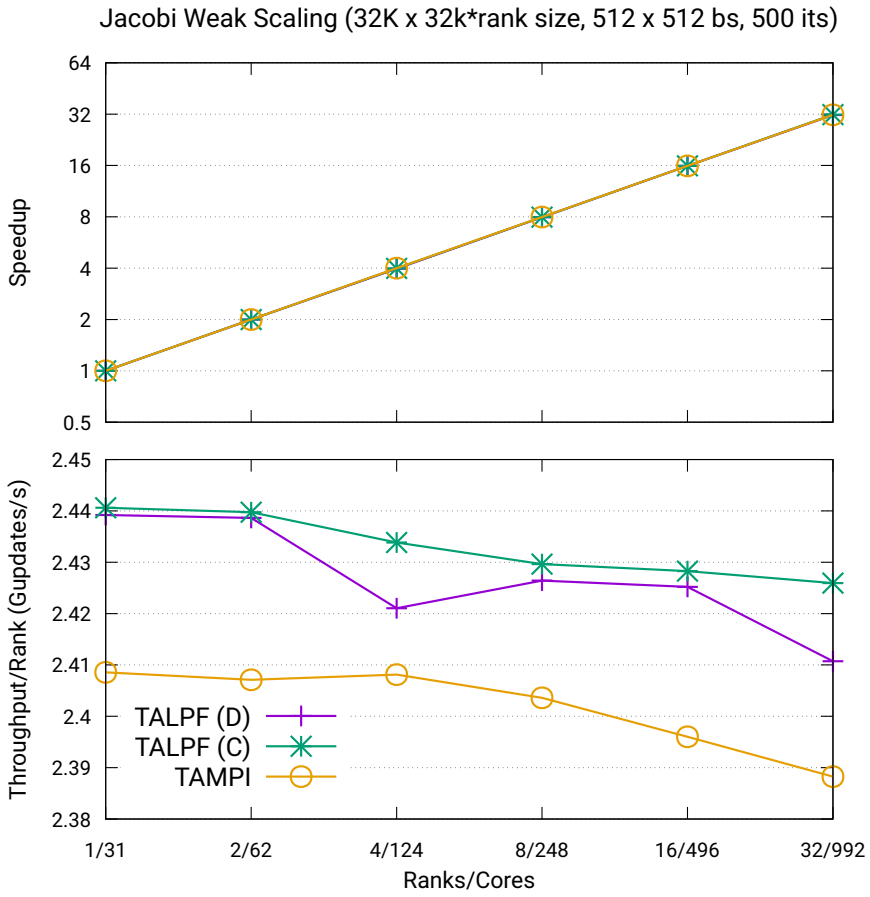


Fig. 7.3: Jacobi Weak Scalability test, showing the speedup (top) and the normalized throughput (bottom). The Purple line is the TALPF(D) version, the green line is the TALPF(C) version, and the yellow line is the TAMPI version.

Figure 7.4 shows an execution trace with TALPF, using TALPF_SYNC_DEFAULT mode. We can observe that the first iteration, after the warmup iteration, is bigger than the rest, and the final iterations are smaller. That is because, with the Jacobi solver, there is enough freedom for the compute tasks that can advance iterations.

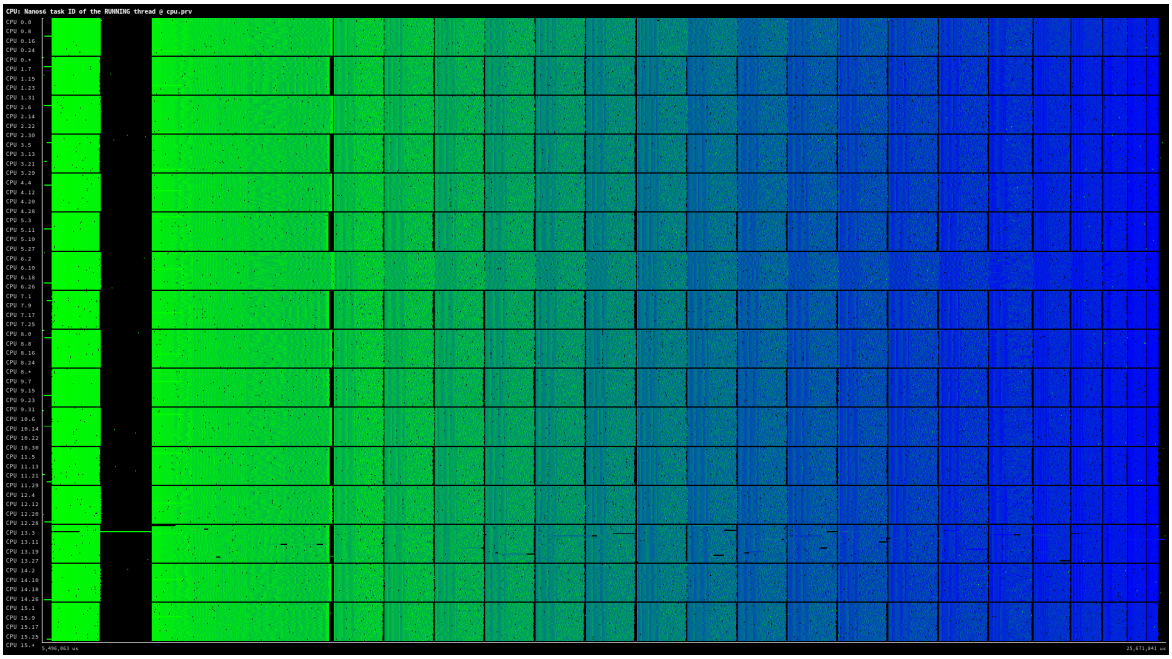


Fig. 7.4: Jacobi trace of a TALPF(D) version execution performing 20 iterations (plus an initial warmup iteration). Task execution is printed in color and the idle/runtime time is shown in black. The view shows the task identifier coded in a gradient that goes from light green (first tasks generated at the beginning) to dark blue (last tasks at the end).

7.3 Heat-Diffusion: Gauss-Seidel

The Gauss-Seidel [20] approach for the heat-diffusion problem consists in one matrix that updates itself and follows an order of top-down and left-right, making the updates dependent on the left and top value updates, this cause a pattern that is harder to parallelize in comparison to the Jacobi.

This computation pattern and the lack of double buffering make this problem hard to make a hybrid version with TALPF, causing a lot of performance losses. As Figure 7.5 shows, in a BSP approach, the coarse synchronization approach does not allow the adjacent ranks to work in parallel causing the ranks to be half of the time doing nothing.

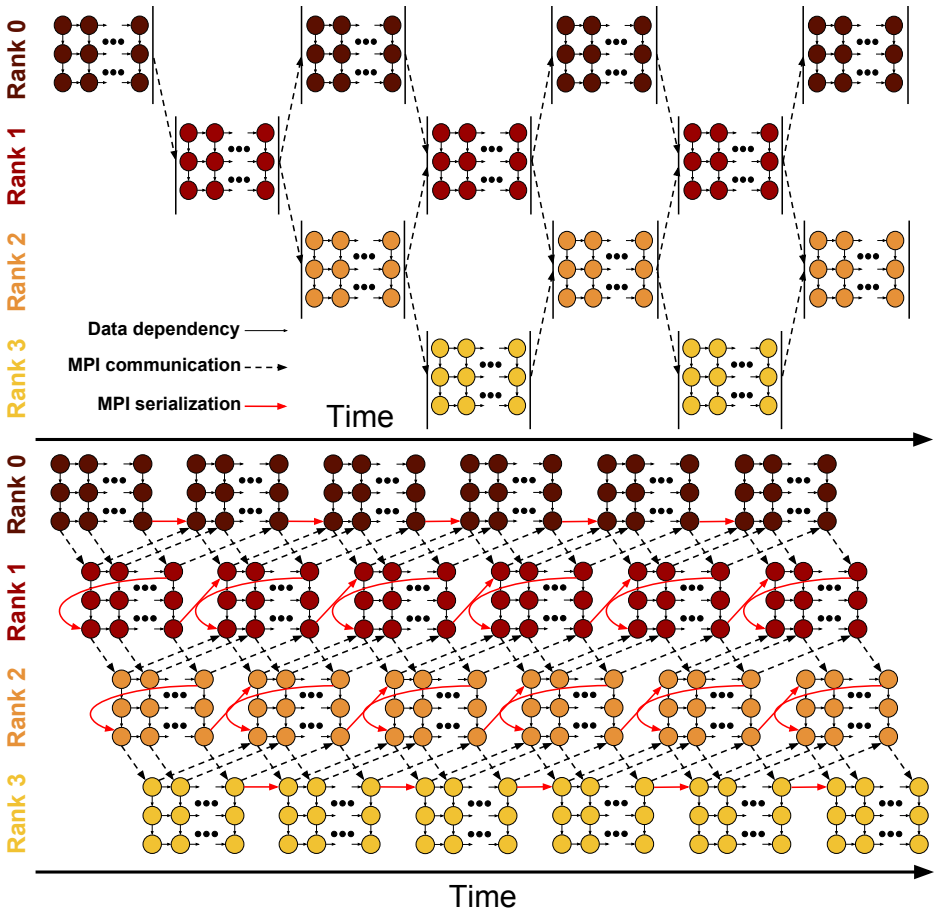


Fig. 7.5: Comparision between a BSP/fork-join approach (top) and MPI data-flow approach (bottom).

Figure 7.6 shows an extreme case of the problem mentioned in Figure 7.5 with a real execution of TALPF. This example is an execution of two iterations with eight ranks, these two iterations took ten iterations in total, whereas in the best case, only two ranks were executed. This is an extreme case to illustrates the problem, in a normal execution with several iterations, only half of the ranks will be executing at the same time, and there is an initialization/finalization phase, where only some ranks can be executed.

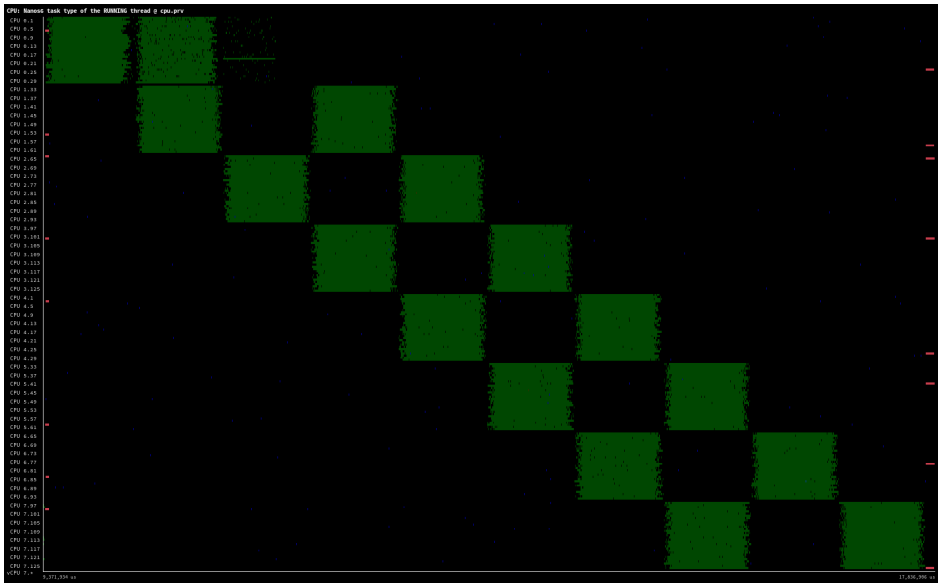


Fig. 7.6: Trace of an extreme TALPF Gauss-Seidel case, with eight ranks and two iterations.

Figure 7.7 shows the Strong Scalability test, we can observe that while TAMPI maintains good scalability, TALPF loses half the normalized throughput of the base case (without any communication), and over eight ranks loses even more throughput. As we will see in the Weak Scalability test, this loss of performance over eight ranks is due to a lack of parallelism of the problem size itself.

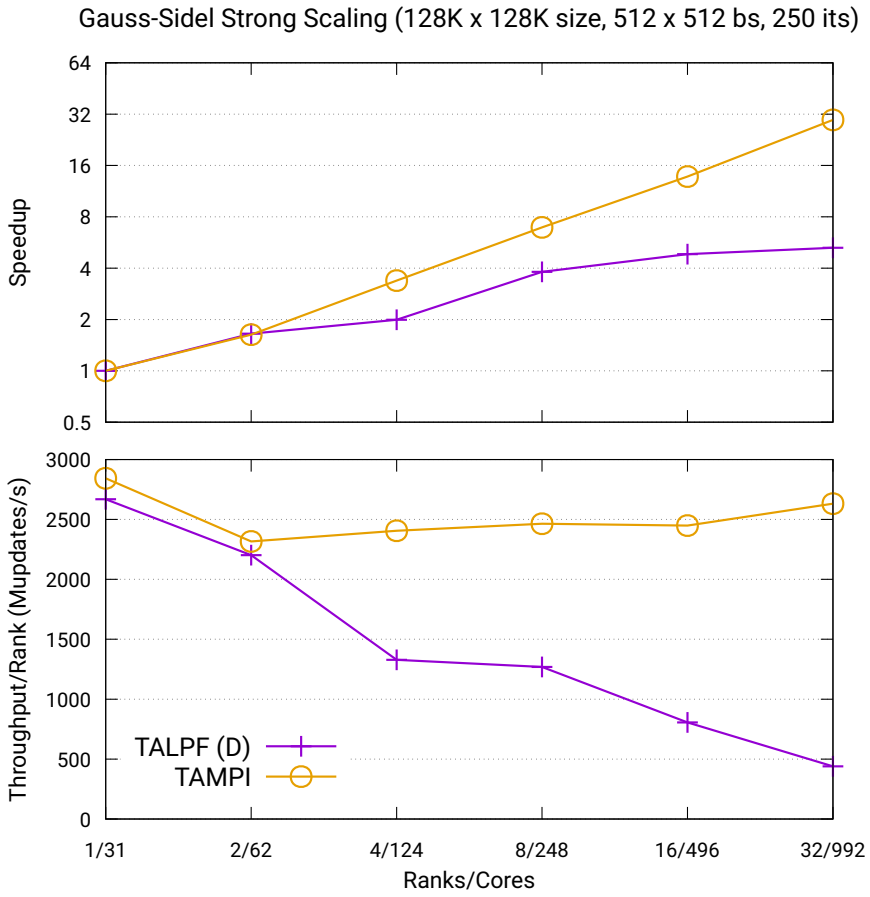


Fig. 7.7: Gauss-Seidel Strong Scalability test, showing the speedup (top) and the normalized throughput (bottom). The Purple line is the TALPF(D) version and the yellow line is the TAMPI version.

Figure 7.8 shows the Weak Scalability test, in this test we can observe that the TAMPI version does not have good weak scalability, while in the TALPF version, we can observe that the minimum normalized throughput is around half of the base case, which is what we expect as only half of the ranks are working at the same time.

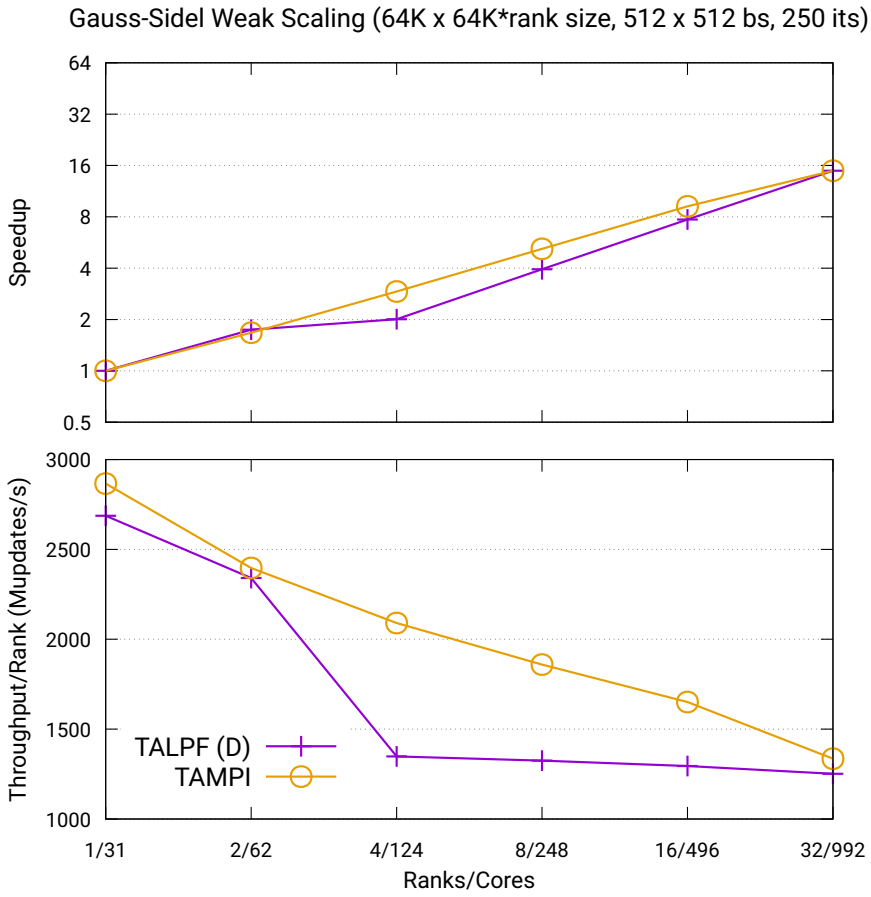


Fig. 7.8: Gauss-Seidel Weak Scalability test, showing the speedup (top) and the normalized throughput (bottom). The Purple line is the TALPF(D) version and the yellow line is the TAMPI version.

7.4 Nbody

The Nbody simulation is an application that calculates the forces between particles, then updates the position. These force calculations grow quadratic, as every particle force has to be calculated with all other particles.

As each particle has to be calculated against all the other particles, this creates a dependency over the force value making a chain for each particle. As we will see this chain of tasks can impact the performance as *Nanos6* prioritizes these chained tasks to execute one after the other.

In the case of hybrid mode, the particles are divided between all the ranks, and the particles are exchanged in a ring pattern, where the particles are sent to the next rank in an auxiliary memory, then the rank calculates the forces. This process is repeated until all ranks have seen all the particles. This ring pattern is not a bidirectional

communication so it will cause a programmability overhead in the case of the zero-cost synchronization modes.

In this benchmark, we tried two versions, the first one, with all the default values but without the immediate successor of *Nanos6*, and the second one, with the same parameters, and we added priority to the computing task to break the chains.

In figure 7.9 we can observe the Strong Scalability test of the first version, where all the versions have a similar performance with four or fewer ranks, but above 4 ranks TALPF is losing performance faster than TAMPI. Also, the TALPF with the DEFAULT mode and the CACHED with BARRIER mode is significantly slower than the CACHED with the ACK version, proving that global synchronization deteriorates performance, especially with a higher number of ranks.

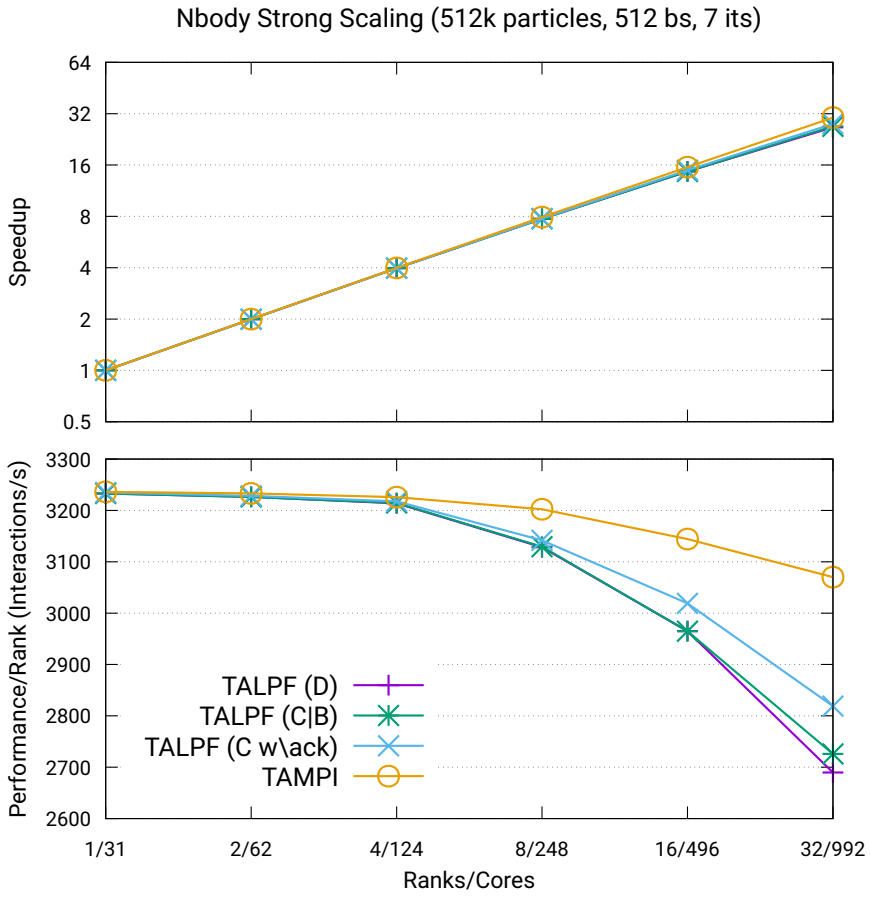


Fig. 7.9: Nbody Strong Scalability test, showing the speedup (top) and the normalized performance (bottom). The Purple line is the TALPF(D) version, the green line is the TALPF(C | B) version, the blue line is the TALPF(C w/ ACK) version, and the yellow line is the TAMPI version.

Figure 7.10 shows the Weak Scalability test of the first version, where to avoid the quadratic growth we only duplicate the size with 4 and 16 nodes. This test shows

something very similar to the previous one, TAMPI is the fastest and the one which escalates better, followed by TALPF with the LPF_SYNC_CACHED mode with an ACK message, followed by the two other TALPF versions which had a barrier.

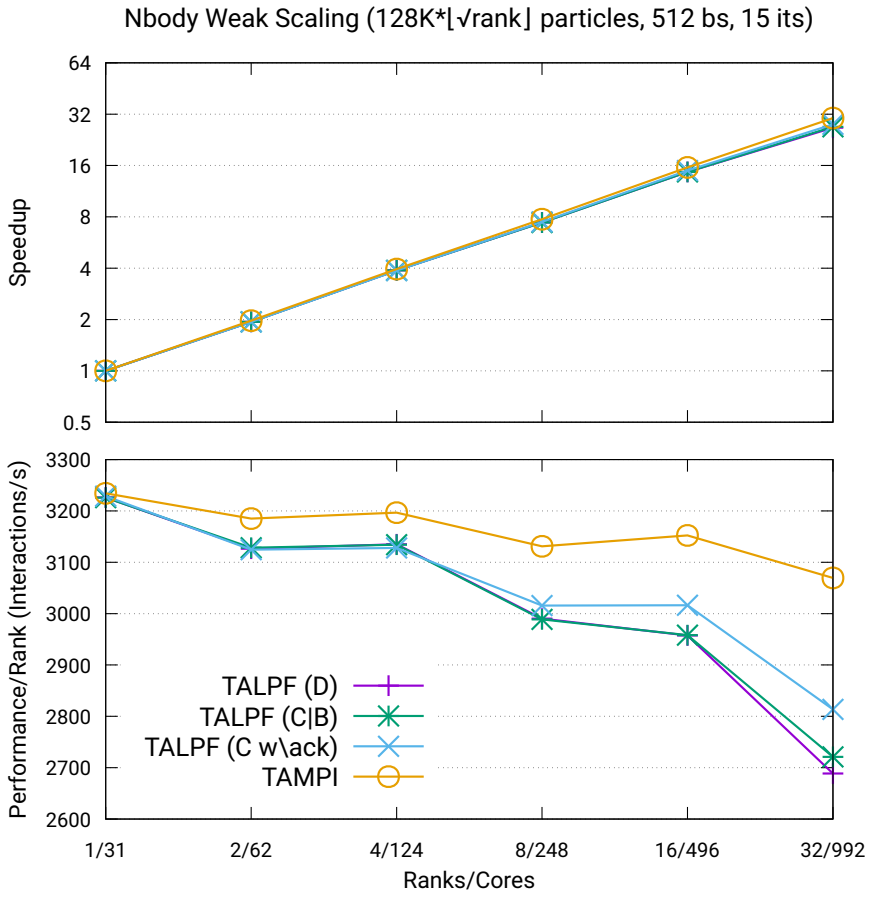


Fig. 7.10: Nbody Weak Scalability test, showing the speedup (top) and the normalized performance (bottom). The Purple line is the TALPF(D) version, the green line is the TALPF(C | B) version, the blue line is the TALPF(C w/ ACK) version, and the yellow line is the TAMPI version.

In figure 7.11 we can observe what happens in an iteration of TALPF with the LPF_SYNC_CACHED mode with an ACK message, even if we remove the immediate successor, there are still some chains that stick out of the other tasks. To solve we added a priority to make sure the tasks are executed to break all the chains.

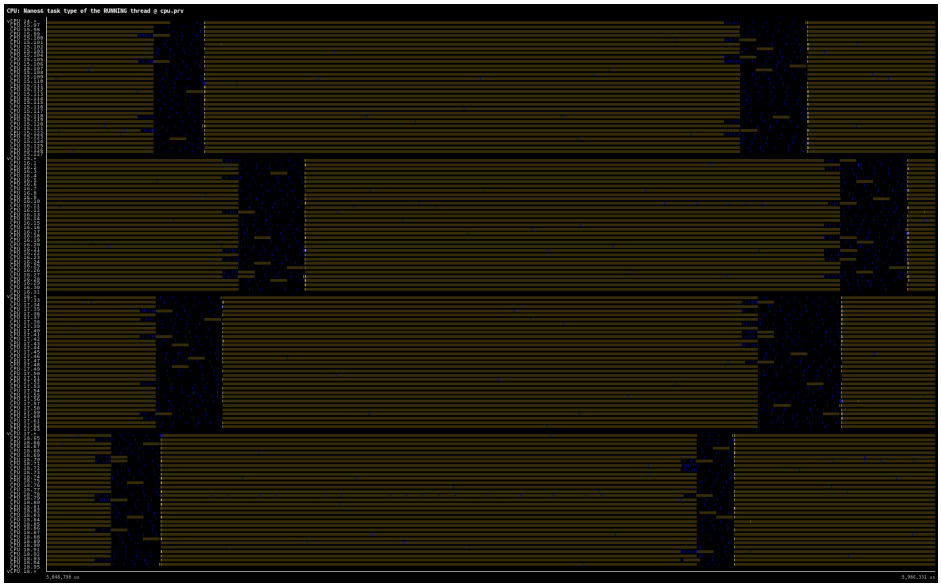


Fig. 7.11: Nbody trace of a TALPF(C w/ ACK) version with 32 ranks. Zoom in four ranks and one iteration.

In Figure 7.12 we can observe the new performance with the priorities. Now the TALPF versions reach a similar performance to TAMPI. We also added a TALPF version with talpf_get and the LPF_SYNC_DEFAULT mode to see if there is a difference in performance, in this case, the performance seems identical to the talpf_put.

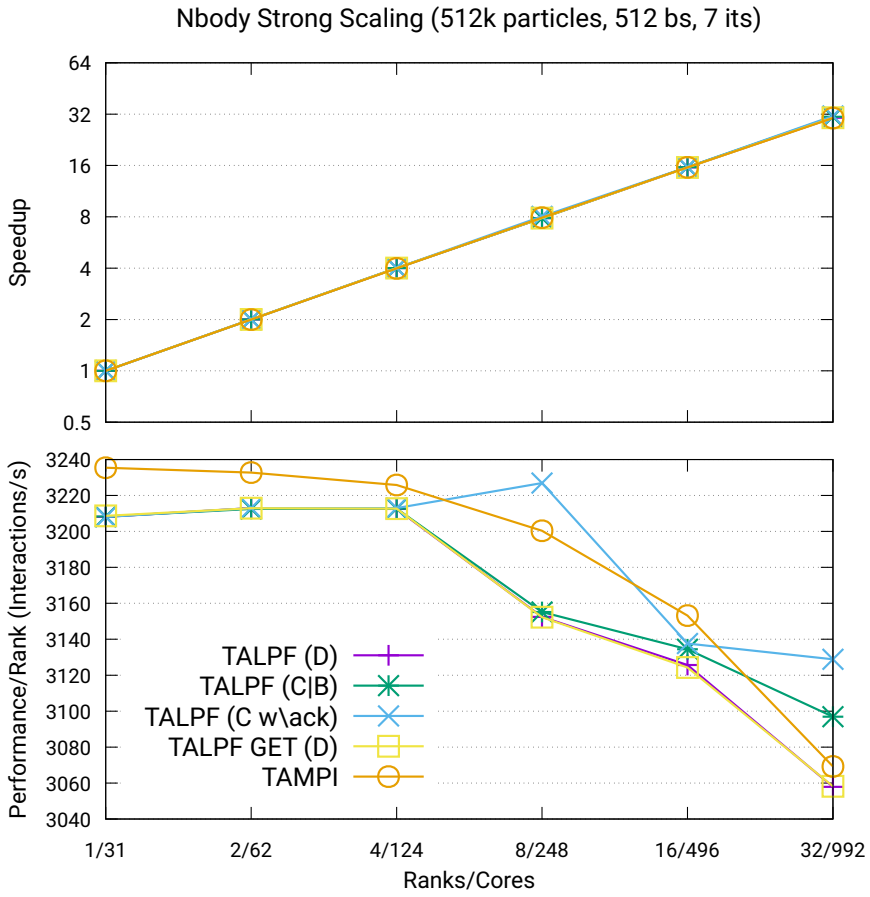


Fig. 7.12: Nbody with priorities Strong Scalability test, showing the speedup (top) and the normalized performance (bottom). The Purple line is TALPF(D) version, the green line is TALPF(C | B) version, the blue line is TALPF(C w/ ACK) version, the yellow line is TALPF GET(D) version, and the orange line is TAMPI version.

Figure 7.13 shows similar results for the Weak Scalability test, where all the TALPF versions have improved their scalability, and now have a similar scalability with the TAMPI version. Again the talpf_get version has a similar performance as the talpf_put.

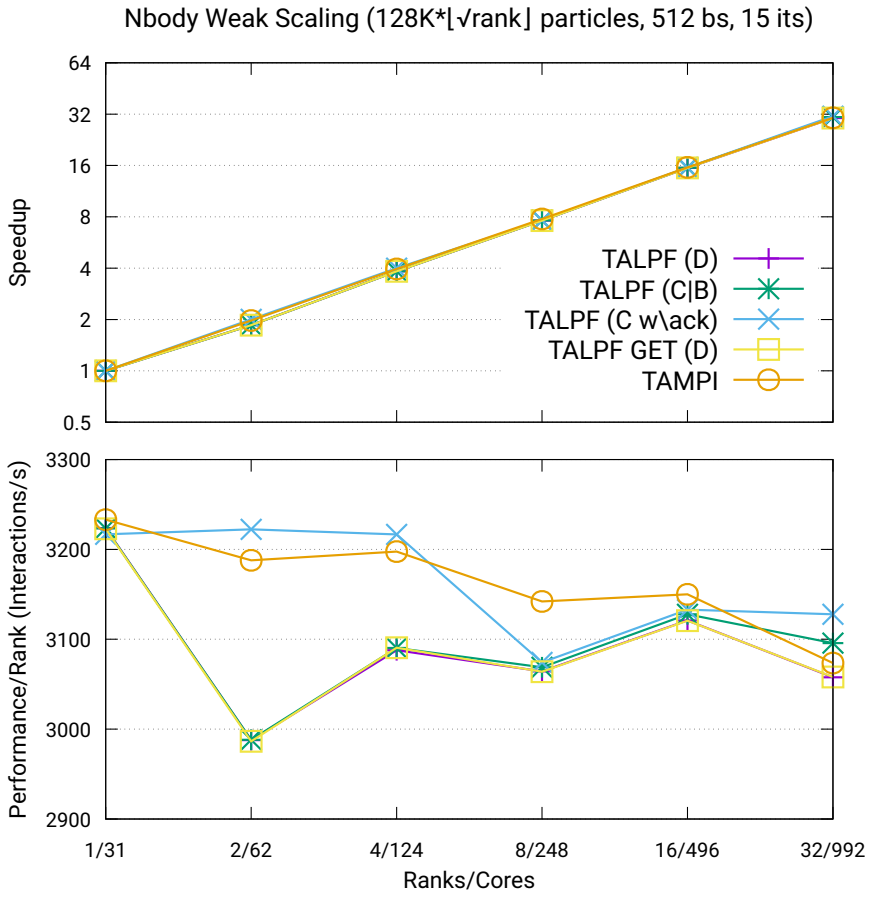


Fig. 7.13: Nbody with priorities Weak Scalability test, showing the speedup (top) and the normalized performance (bottom). The Purple line is TALPF(D) version, the green line is TALPF(C | B) version, the blue line is TALPF(C w/ ACK) version, the yellow line is TALPF GET(D) version, and the orange line is TAMPI version.

In Figure 7.14 we can observe what happened to the iteration, now there is no chain sticking out, only a task that it could not fit with the others, probably because of the topology of the cores, where the size of the problem is a power of two but there are only 31 cores.

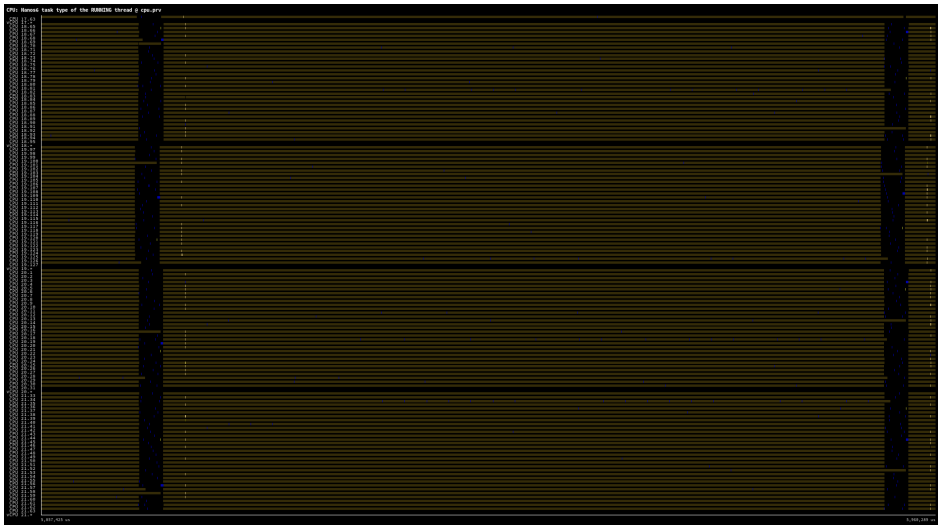


Fig. 7.14: Nbody with priorities trace of a TALPF(C w/ ACK) version with 32 ranks. Zoom in four ranks and one iteration.

7.5 Matrix Multiplication

The Matrix Multiplication benchmark consists of an iterative Matrix Multiplication. The Matrix Multiplication follows the following equation $C[i][j] = \sum_{k=0}^{N} A[i][k] * B[k][j]$ for all positions of the matrix, that means it is a cubic problem, which as we will see it makes difficult to get a size that fits all the range of parallelization, and also makes the weak scalability harder to grow.

In the hybrid version, every rank has its slice of the matrix, and like the Nbody simulation, the communication pattern is a ring, where in every substep, a section of the matrix is passed to the next rank. There is a problem in this approach that we did not have in the Nbody benchmark. In this case, we are exchanging bigger messages, and as we will see in the Strong Scalability test and Weak Scalability test, this will cause an oversubscription to TAMPI, dampening its performance.

Figure 7.15 shows the Sreong Scalability test of the Matrix Multiplication benchmark, we can observe that all the versions perform similarly except TAMPI which loses performance faster than the TALPF versions. The loss of performance of TAMPI is due to the oversubscription of the communication. We can also observe that all the versions lose around a third of their normalized performance using 32 ranks, that is because the size of the problem does not allow enough parallelism.

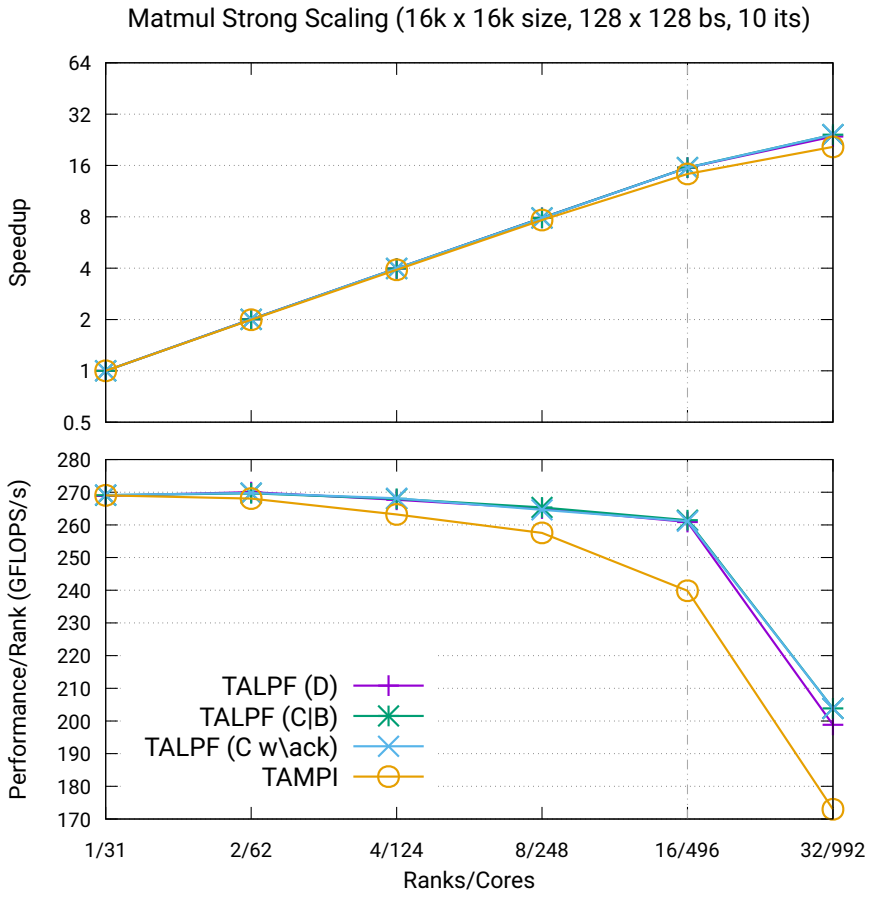


Fig. 7.15: Matrix Multiplication Strong Scalability test, showing the speedup (top) and the normalized performance (bottom). The Purple line is the TALPF(D) version, the green line is the TALPF(C | B) version, the blue line is the TALPF(C w/ ACK) version, and the yellow line is the TAMPI version.

Figure 7.16 shows the Weak Scalability test of the Matrix Multiplication benchmark, where the size only is duplicated after (included) the eight ranks version, to avoid the cubic growth. The Weak Scalability test shows similar results as the Strong Scalability test, which makes sense as most of the samples as the size does not increase, and therefore behaves like a Strong Scalability test.

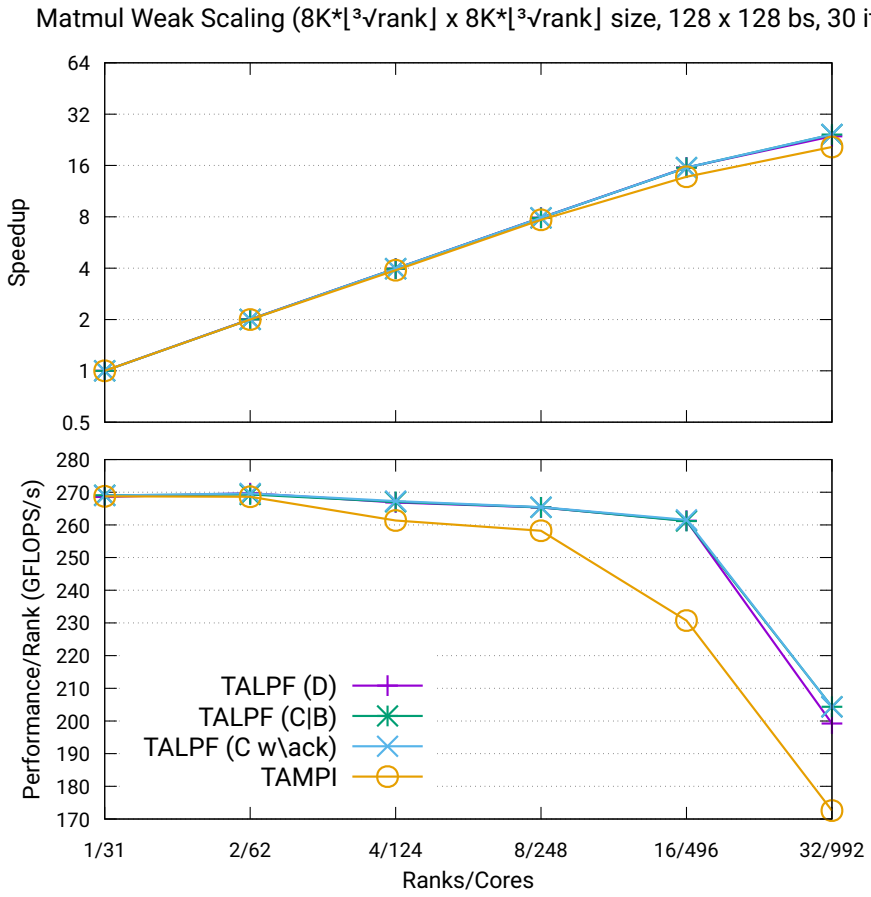


Fig. 7.16: Matrix Multiplication Weak Scalability test, showing the speedup (top) and the normalized performance (bottom). The Purple line is the TALPF(D) version, the green line is the TALPF(C | B) version, the blue line is the TALPF(C w/ ACK) version, and the yellow line is the TAMPI version.

Figure 7.17 shows a trace of four ranks in the first and second iterations of execution of the Matrix Multiplication benchmark with the same parameters of the Strong Scalability test with 32 ranks. In each iteration every rank can only execute 16 compute tasks in parallel due to the dependencies of the tasks, as we can observe there is a clear overlap between the first and second iterations, also the second and third iterations, where all the cores of the ranks are being used. That explains why in the Strong Scalability test and Weak Scalability test the TALPF versions lost only a third of the normalized performance instead of a half.

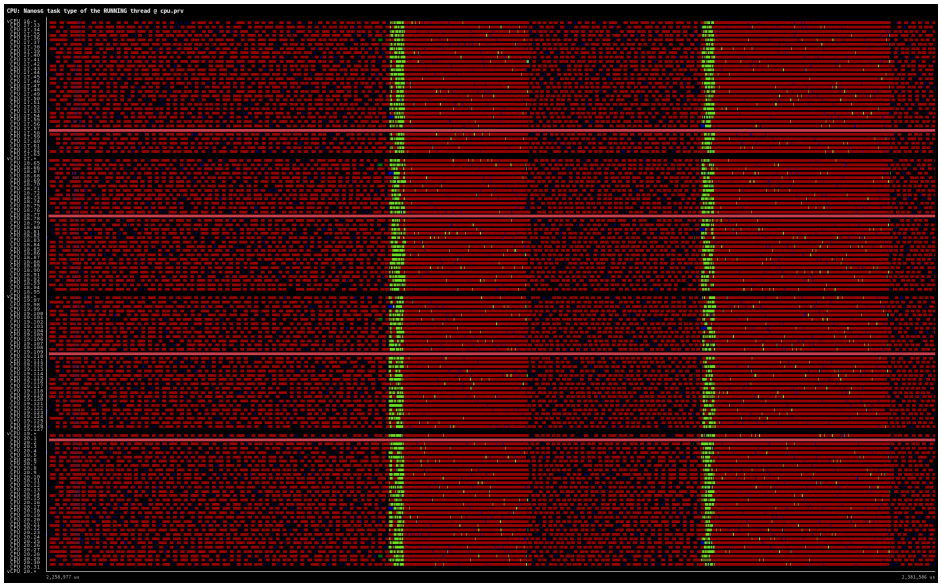


Fig. 7.17: Matrix Multiplication trace of a TALPF(C w/ ACK) version with 32 ranks. Zoom in four ranks and the first and second iterations. The red tasks are compute tasks, the green tasks are memcopies, and the blue tasks are the polling tasks.

8 | Conclusion

In this project we have introduced a combination of a BSP model with a data-flow model to exploit parallelism at an intra-node level and an inter-node level, maintaining the simplicity of the BSP model and the performance of a data-flow model. This new combination allows us to overlap the computation and communication phases optimally using all the available resources. Also, thanks to the data-flow model flexibility, we are able to delay or advance some computation tasks that are not strongly linked to the BSP model phases.

To further optimize the BSP model, we designed new zero-cost synchronization methods, that allow the synchronization to be more local, and thus, improve the performance and the flexibility of our applications. As we mentioned, these new methods only have two drawbacks. Firstly, they only work in a repetitive environment or can be defined by the user which limits the programmability. Secondly, bi-directional communication is required, forcing the applications without bi-directional communication to add empty messages to ensure bi-directionality, limiting the programmability. The second limitation could be easily solved using a new synchronization method which is left as future work (Chapter 9).

We have evaluated our implementation against TAMPI, to prove that we can hide all the synchronization overhead in one coarse primitive without losing performance. The results showed us that, while the modes with a barrier lost performance with a large number of processes, the new zero-cost synchronization methods have a similar performance to TAMPI, proving that the coarse synchronization is enough in the applications with iterative behavior.

9 | Future Work

As we have seen in Chapter 7, our implementation achieves a competitive performance against TAMPI with the new modes. TALPF has two main programmability drawbacks, the application needs double buffering, at least if the user uses the `talpf_put` primitive, and the need for the bidirectional communication to correctly synchronize two processes in the new modes.

To address the issue of bidirectional communication, we introduced a synchronization message that ensures proper synchronization between processes. This message needs to be manually added, but to enhance programmability, it could be encapsulated within the synchronization primitive, eliminating the need for additional programming overhead.

To overcome the double buffering problem, we could use a technique where LPF internally manages two memory regions: one for the user to interact with and another for LPF's communication operations. These changes in the memory regions will be merged during the synchronization process. This approach can improve programmability, but it could also decrease performance.

Another path that can follow the project is to implement a TALPF-like approach with TAMPI using the MPI one-sided primitives, this will give us the opportunity to do a more fair comparison, as now both libraries use different approaches. TALPF is one-sided while TAMPI is two-sided, and TALPF uses a coarse synchronization while TAMPI uses a fine-grained synchronization.

Bibliography

[1] R. Rabenseifner and G. Wellein. Comparison of parallel programming models on clusters of SMP nodes. In *Proceedings of the International Conference on High Performance Scientifc Computing*, March 2003. 1

[2] Rolf Rabenseifner. Hybrid parallel programming: Performance problems and chances. In *45th Cray User Group Conference*, pages 12–16, 2003.

[3] Gabriele Jost, Haoqiang Jin, and Ferhat F. Hatay. Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster. Technical report, NASA, September 2003. 1

[4] MPI Forum. MPI: A Message-Passing Interface Standard. Version 4.0, June 9th 2021. Available at: https://www.mpi-forum.org/docs/. 1, 5

[5] GASPI Forum. GASPI: Global Address Space Programming Interface. Version 17.1, February 7th 2017. Available at: https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-17.1.pdf. 1

[6] OpenMP Architecture Review Board. OpenMP Application Programming Interface. Version 5.2, November 2021. Available at: https://www.openmp.org/. 1

[7] Barcelona Supercomputing Center. OmpSs-2 Specification, . URL https://pm.bsc.es/ompss-2-docs/spec/. 1, 25

[8] Wijnand Suijlen and AN Yzelman. Lightweight parallel foundations: a model-compliant communication layer. *arXiv preprint arXiv:1906.03196*, 2019. 1, 19

[9] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 5–16, 2010. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810091. 2, 31

[10] Kevin Sala, Xavier Teruel, Josep M Perez, Antonio J Peña, Vicenç Beltran, and Jesus Labarta. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Computing*, 85:153–166, 2019. 2, 31

[11] Kevin Sala, Sandra Macia, and Vicenç Beltran. Combining one-sided communications with task-based programming models. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 528–541. IEEE, 2021. 2, 33

[12] Barcelona Supercomputing Center. Task-Aware GASPI (TAGASPI) Library, . URL https://github.com/bsc-pm/tagaspi. 2, 33

[13] Barcelona Supercomputing Center. Task-Aware MPI (TAMPI) Library, . URL https://github.com/bsc-pm/tampi. 31

[14] Barcelona Supercomputing Center. Mercurium Compiler, . URL https://github.com/bsc-pm/mcxx. 34

[15] Barcelona Supercomputing Center. Nanos6 Runtime System Library, . URL https://github.com/bsc-pm/nanos6. 36

[16] Argonne National Laboratory. MPICH. URL https://www.mpich.org. 37

[17] Barcelona Supercomputing Center. ovni, . URL https://ovni.readthedocs.io/en/master. 37

[18] Barcelona Supercomputing Center. Paraver, . URL https://tools.bsc.es/paraver. Accessed: 2018-05-21. 38

[19] Giampiero Esposito. *The Heat Equation*, pages 329–334. Springer International Publishing, Cham, 2017. ISBN 978-3-319-57544-5. doi: 10.1007/978-3-319-57544-5_23. URL https://doi.org/10.1007/978-3-319-57544-5_23. 62

[20] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. ISBN 0-89871-396-X. 66