# HiCR, an Abstract Model for Distributed Heterogeneous Programming

Anonymous Author(s)

## Abstract

We present HiCR, a model to represent the semantics of distributed heterogeneous applications, frameworks, and runtime systems. The model describes a minimal set of abstract operations to enable hardware topology discovery, kernel execution, memory management, communication, and instance management, without prescribing any implementation decisions. The goal of the model is to enable execution in current and future distributed heterogeneous systems without the need for significant refactoring, while also being able to serve any governing parallel programming paradigm. We explain how the model's components and operations are realized by a plugin-based approach that takes care of device-specific implementation details. We present examples of the model's main operations and experimentally demonstrate how they run equally on a diversity of platforms.

## CCS Concepts

• **Computing methodologies** → **Distributed computing methodologies**; *Parallel computing methodologies.*

## Keywords

Heterogeneous Computing, Distributed Computing, Runtime Systems

**ACM Reference Format:**

Anonymous Author(s). 2025. HiCR, an Abstract Model for Distributed Heterogeneous Programming. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC25).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Recent advancements in artificial intelligence algorithms, especially those of Large Language Models (LLMs), present substantial demands from modern computing systems. These models require computational power beyond that of traditional CPUs. In some instances, their memory and compute requirements go beyond what a single device or node can offer. This trend has driven the rise of *distributed heterogeneous systems* as a leading approach for executing AI pipelines. These systems are characterized by, first, the use of *accelerators* such as Graphics and Neural Processing Units (GPUs and NPUs) for higher-dimensional tensor operations, and second, the use of distributed computing to scale out computational performance and memory capacity.

Distributed heterogeneous systems induce significant complexities to software development, especially in applications that strive to maximize performance and fully exploit the hardware and interconnect capabilities. On one hand, the use of accelerators often requires the use of vendor-specific interfaces, resulting in tightly coupled implementations. On the other hand, distributed applications must deal with deployment-specific requirements, such as those for cloud platforms, data centers, or smartphones.

The complexity of handling devices and interconnect-specific technologies undermines the application's portability, as changes to the underlying hardware often necessitate extensive refactoring, even when semantics remain the same. These challenges highlight the need for flexible systems capable of hiding platform-specific implementation details and seamlessly adapt to new hardware and technologies.

We present HiCR (pronounced as '*hiker*'), a model to represent the semantics of distributed heterogeneous applications. The model prescribes a minimal set of objects and operations to enable hardware topology discovery, kernel execution, memory management, communication, and instance management. These operations are not tied to any given technology. Instead, they describe an application in terms of abstract operations that do not prescribe implementation details. As a result, any HiCR-based application is able to reach its intended result regardless of the hardware provided.

The HiCR model employs a plugin-based approach, delegating to third-party developers the responsibility of translating its semantics into implementation-specific directives. The benefits of this design are twofold. First, applications need to be written only once and can then seamlessly execute across a wide range of technologies and platforms. Second, any newly developed plugin automatically becomes available to all applications, allowing them to benefit from the added functionality without further modifications.

The contributions of this paper are: (a) the introduction of the *Runtime Support Library*, as an intermediate layer between an application and the underlying technologies; (b) the proposal of HiCR, as an abstract model for the operations that this layer should offer, and (c) an open-source implementation of the model.

The rest of the paper is as follows: in §2 we introduce the HiCR model; in §3, we describe its implementation and highlight the key components that enables application portability; in §4, we show empirical results that demonstrate how HiCR applications obtain equal results with different technologies without the need of refactoring; in §5, we discuss related work, and in §6 we provide final thoughts and discuss future work.



*SC25, St. Louis, MO*



ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

## 2 The HiCR Model

The HiCR model exposes a minimal set of building blocks to describe the semantics of an application in any von Neumann-like distributed computing system. That is, it assumes that a computing system contains a set of processing units connected to contiguous random access memory spaces. It also allows the discovery and use of the components of any heterogeneous system at runtime, and the deployment of an application on diverse (distributed) environments, such as mobile, cloud, and datacenter.

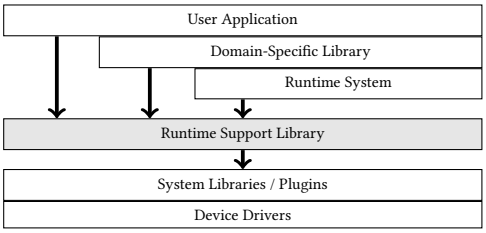


**Figure 1: A layered view of runtime support for user applications. Any of the layers may invoke functionality of one or more of the layers below it to support runtime operations. We propose HiCR as a model for a *Runtime Support Library* layer to abstract common operations and allow the user to express clear semantics without implementation considerations.**

Fig. 1 represents the software stack on which a user application may operate at runtime. Any of these layers may rely on functionality provided by one or multiple layers below it. For example, a given application may be programmed in such a way to only require access to device drivers (low-level interrupts) or system libraries (e.g., *MPI* [1]). These layers are typically accessed by expert programmers with a deep knowledge of such technologies. Other users may prefer higher productivity at the expense of precise execution control and opt for domain-specific libraries. Intermediate users may prefer a programming model that delegates the complexities of device access and scheduling execution to an underlying runtime system.

Runtime systems can automatically resolve communication operations and device management directly, encapsulating the required accesses to the corresponding low-level libraries. For example, the *StarPU* [2] runtime system interfaces directly with MPI and *CUDA* [3] for distributed communication and GPU operations, respectively. The limitation with this approach is that applications cannot be easily ported to employ other existing or future technologies if the underlying runtime system and its API are not updated; while even if so, the application itself may require significant refactoring.

To solve the issue of portability, we propose the use of a *Runtime Support Library* layer to serve as an interface between application code and the underlying libraries. This layer provides an implementation-agnostic API consisting of runtime computation, communication, and system management building blocks. We propose HiCR as a model to describe such building blocks.

HiCR has no semantic prescriptions. Instead, the operations in the API can be employed independently by any programming model and software layering running on top of it (e.g., a tasking runtime system or an SPMD distributed application). By decoupling the runtime support library from the programming model, we enable HiCR to be used by highly diverse programming approaches. Any runtime system, domain-specific library or user application may delegate all low-level operations to HiCR without the need to consider implementation details. These details are later resolved by device-specific implementations, or *plugins*, of the HiCR model. In this way, the application may be ported to other systems and technologies simply by selecting a different set of plugins.

### 2.1 Model Description

Fig. 2 shows HiCR's components and the operations defined among them. The model components are divided into three groups: *managers*, *stateless* and, *stateful*. Managers are components whose operations have an effect on the system. For example, they can trigger the execution of a kernel, the copying of data from one device to another, or create a new application instance. In addition, only managers can create instances of other, stateless and stateful components. Stateless components represent information about the system or the static description of a function. As such, these components can be copied, replicated, serialized, and transmitted as required. On the other hand, stateful components represent objects with a finite lifetime whose internal state is subject to change. For example, a running thread or a GPU stream are stateful objects that may be, at any point in the application's time, executing, suspended, or finalized. These components are unique and therefore cannot be replicated.

*2.1.1 Instance Management.* The model defines an *Instance* as any subset of the entire (multi-node) system's available hardware elements, capable of executing independently. An instance is typically implemented as an OS process with full or partial access to a node's CPU, memory, and accelerator devices. The model requires that no two running instances share access to the same devices. Being disjoint, the only contact point for any two instances is via distributed memory communication.

All operations involving instances are handled by the *Instance Manager*. The user may use one of, or a combination of, two ways in which the instance manager enables distributing execution. The first is to detect already created instances. This is typically in cases where the underlying library (e.g., MPI) launches all instances at launch-time and the instance manager allows retrieving them as a list. The second way is to create new instances during runtime. In this case, the instance manager, running on the initial instance, creates new instances on remote servers at runtime.

Creating a new instance requires passing an *Instance Template* object. This object encapsulates the description of a required topology. If the underlying system library or cloud management platform supports it, this topology will indicate the minimal hardware resources to allocate to the new instance.

Each running instance is semantically equivalent to every other, although only one of them is considered a root instance. A *Root Instance* is either the first instance to be created, or one within the first group of instances created at launch time. The sole purpose of designating an instance as root is to provide a tie-breaking mechanism.

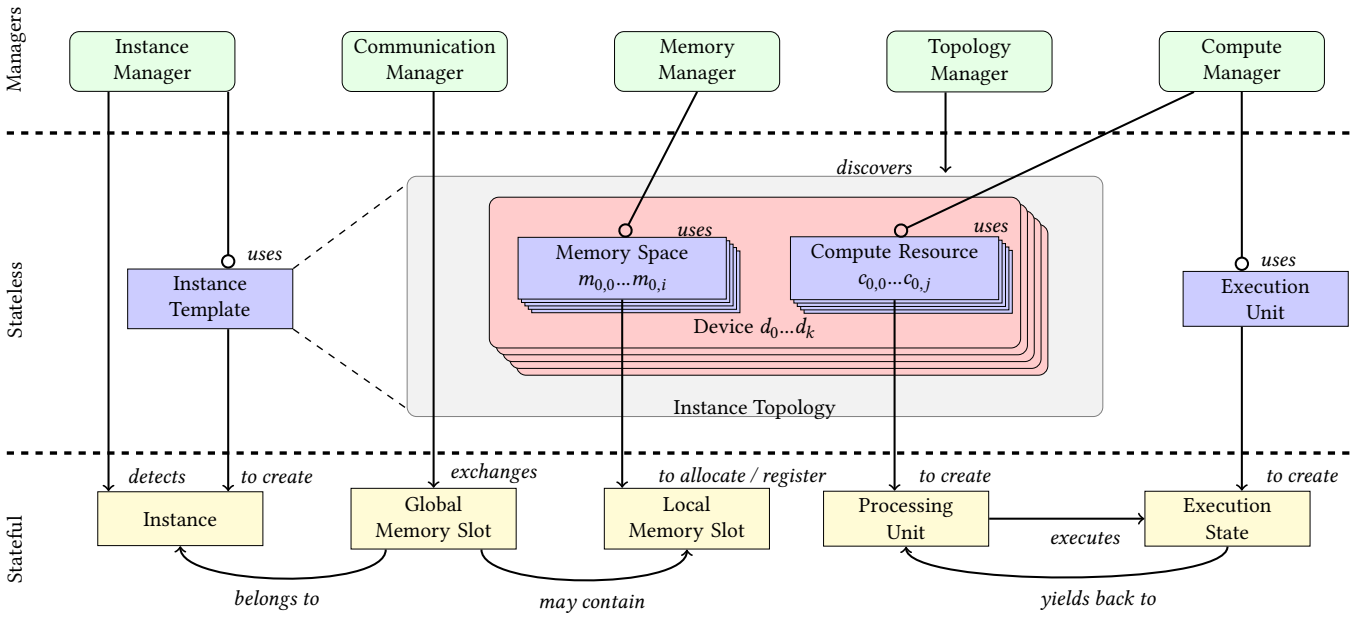


**Figure 2: Diagram showing the components of the HiCR model and the available operations between them. The model is divided in three component groups: *Managers*, components whose operations represent an application's semantic building blocks; *Stateless*, components that represent static information, and *Stateful*, components with an internal state that mutates over time.**

*2.1.2 Topology Management.* A *Topology* represents a full or partial information of an instance's available hardware devices. It comprises a set of *Devices*, a representation of a single hardware element (e.g., NUMA Domain, or GPU), containing zero or more memory spaces and compute resources.

A *Memory Space* represents a hardware element that exposes explicitly addressable, contiguous memory segments of non-zero size. Since memory spaces are meant to inform about a device's real memory capacity, the actual physical size is given, and not the size of the virtually addressable space. For example, the system main memory may be exposed as either a single uniform memory access (UMA) memory space (e.g., 128GB), or as multiple non-uniform (NUMA) memory spaces (e.g., 2 x 64GB). For heterogeneous devices, memory spaces may include device RAM, addressable caches, as well as high-bandwidth memories.

A *Compute Resource* represents a hardware or logical element, capable of performing computation. Typical examples of compute resources are CPU cores, each capable of executing a function independently. They can also represent vector and cube cores in an accelerator, capable of executing discrete kernels and streams. This component contains all the information necessary to uniquely identify the corresponding processor.

Topologies are discovered by a *Topology Manager*. A combination of different topology managers, each targeting a specific technology, can be used to gather information of different devices. The resulting topologies can be merged and serialized, allowing users to transmit the full picture of the local hardware information between instances.

*2.1.3 Memory Management.* Memory management in HiCR consists of the creation, exchange and destruction of *Local Memory Slots*. These slots represent the source and destination buffers in data transfers within the scope of a single HiCR instance. They contain the minimum information required to describe a segment of memory (e.g., size, starting address).

The *Memory Manager* object exposes a similar interface to that of the standard C library (i.e., *malloc* and *free*) for the allocation and freeing of local memory slots. However, while the C `malloc` operation defaults to obtaining the allocation from the system's main memory, the HiCR model expands on it, allowing the specification of which memory space to use as source for the allocation. As long as the memory manager recognizes the specified memory space as one in which it can operate and there is enough space in it, the operation will be successful.

The model also allows the registration of a local memory slot based on an existing allocation by specifying its address, size, and the memory space to which it belongs. The memory manager will record the provided information and return a memory slot object that can be used for data transfers. This feature is useful for cases when the user needs to perform a HiCR operation, such as remote communication, on an allocation from an external library (e.g., by a math library).

*2.1.4 Communication Management.* All communication in the model is mediated by the *Communication Manager* via the `memcpy` operation. This operation requires the user to specify the source and destination memory slots, as well as the offsets within them and the size of data to communicate. If the communication manager supports communication between the memory spaces to which each of the memory slots belongs, the operation will be initiated. Otherwise, it will be rejected.

The completion of a memory transfer is not guaranteed after the function call returns. Instead, the communication manager exposes a *fence* mechanism that enables the user to suspend execution until the expected number of incoming and outgoing data transfers have been completed. The communication manager also is in charge of creating and exchanging global memory slots.

*Global Memory Slots* are local memory slots that are made accessible to other HiCR instances and can be used as the source or destination of `memcpy` operations. The exchange operation communicates the necessary metadata for remote instances to reach the associated memory slot.

The exchange of global memory slots is a collective operation: all instances must participate by volunteering zero or more local memory slots. The operation returns as many global memory slots as the total amount of local memory slots exchanged. Each of the resulting global memory slots are uniquely identified by a `tag` and `key` pair, as defined by the user. The `tag` element allows for the differentiation of memory slots communicated in different exchange operations, while the `key` element separates any two memory slots within the same exchange. The model allows recovering the internal local memory slot object from a global memory slot, if the latter belongs to the current instance.

Only three directions are allowed for the `memcpy` operation: *Local-to-Local*, *Local-to-Global*, and *Global-to-Local*. The first entailing two local memory slots and are typically resolved by regular memcpy (e.g., OpenCL's `clEnqueueCopyBuffer`) operations. The latter involve transfers between instances where one-sided operations (e.g., `MPI_Put` and `MPI_Get`) may be used, respectively. The `fence` operation can be used to check for completion in any of these scenarios. On the other hand, *Global-to-Global* operations are not permitted in the HiCR model as it entails the possibility of initiating communication between two instances, neither of which orchestrates the operation.

The model contemplates the existence of heterogeneous systems where hosts and accelerators employ independent interconnects. For example, a data transfer operation may be initiated between a global memory slot representing an allocation in a remote accelerator and a local memory slot in a local accelerator. The communication manager will use the accelerator-specific network to satisfy the request, instead of channeling the transfer through the host network. However, a direct communication between a remote accelerator and a host-based local memory slot can only be performed if the provided communication manager supports it.

*2.1.5 Compute Management.* The *Compute Manager* is in charge of managing the compute resources and carrying out all computing operations within the HiCR model. Its primary goals involve managing the lifetime of processing units, prescribing the format of execution units, and overseeing the execution of execution states.

A *Processing Unit* represents a compute resource that has been initialized and is ready to execute. For example, a processing unit representing a CPU core (compute resource) may be initialized as a POSIX thread with a 1-to-1 binding to that core. Similarly, a processing unit may be represented as a stream context, e.g. GPU accelerators. Upon initialization, the processing unit will keep track of its internal state, i.e. ready, executing, suspended (if supported), or terminated.

An *Execution Unit* is the static description of a function, i.e., a procedure that takes inputs, process them, and produces an output. Examples of execution units are C++ lambda functions, to be executed by a CPU resource, or; compute kernels for GPU accelerators. The semantics of an execution unit are given by the user, following the format prescribed by the compute manager.

An *Execution State* represents the execution lifetime of a particular instance of an execution unit, including all the metadata (e.g., inputs, stack, processor state) required to start, suspend and resume (if supported), and finish its execution.

To carry out the execution, the compute manager first needs to create an execution state of a given execution unit. The application must then assign the execution state to an available processing unit. Upon assignment, the processing unit loads the execution state into the processor (e.g., by performing a context switch), and starts computing it. This computation is carried out asynchronously, allowing the rest of the application to perform other operations simultaneously. The completion of an execution state can be queried either in a blocking or non-blocking fashion, and once the execution reaches an end, the execution state yields back to the caller's context and cannot be re-used.

## 3 Implementation

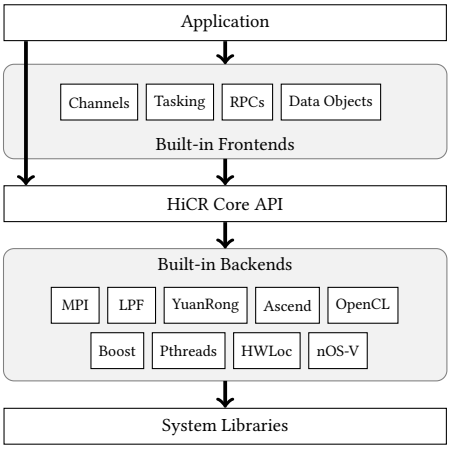


**Figure 3: The current implementation of the HiCR model. Its components and operations are exposed in a *Core API*, which serves as interface between the user-level applications and the underlying system libraries. The core API is distributed together with a set of built-in *backends*, plugins containing the implementation of subsets of model's components for several popular libraries, and *frontends*, HiCR-based libraries providing common, higher-level functionalities.**

We implemented the HiCR model into an open-source, publicly available C++-based core API[1]. The API is distributed together with a set of built-in backend and frontend modules. Fig. 3 shows how these components serve to relate the user application code to the underlying system libraries.

*Backends* are plugin libraries that translate a subset of the HiCR model into implementation-specific operations that underlying system libraries and device drivers can understand. These details are hidden behind HiCR's implementation-agnostic API, allowing a

[1] [Hidden]

HiCR-based program to preserve its semantics across a diversity of devices. Backends are not part of the HiCR core library. Instead, they are included in, or excluded from, the user application as required during compilation.

*Frontends* are C++ libraries that contain higher-level out-of-the-box functionality that may be useful for a wide range of applications. The goal of frontends is to facilitate the adoption of HiCR, minimizing the need for users to use its low-level core API directly by providing ready-to-use features such as communication mechanisms, distributed deployment, and topology discovery. Since these libraries are reliant exclusively on calls to the HiCR Core API, they remain implementation-agnostic and their operations can be supported by different backends.

## 3.1 Programming wih HiCR

To enable the execution of a HiCR-based application, the user code or runtime system need to select the appropriate backends to support the operations therein. This choice is based on the application's own requirements and the system resources available. All the components of the HiCR model are implemented as C++ abstract classes in the Core API. As a result, they cannot be instantiated directly. Instead, each of the backends derives them into complete classes, by resolving implementation-specific details of the operations and fields prescribed by the model.

Fig. 4 shows an example of how some of the built-in backends can be instantiated. The example starts by initializing the MPI library and passing an MPI communicator object to the MPI instance manager constructor (Line 3). Then, it instantiates the *HWLoc* topology and memory managers (Lines 6-9), which requires passing of an `hwloc_topology_t` object as argument. The example then obtains the system's CPU topology, as detectable by HWLoc, into a HiCR topology object (Line 12). Finally, it instantiates the *Pthread*-based communication and compute managers (Lines 15 and 16).

```
1  // Creating MPI instance manager
2  MPI_Init(&argc, &argv);
3  HiCR::MPI::InstanceManager im(MPI_COMM_WORLD);
4
5  // Creating HWLoc topology and memory managers
6  hwloc_topology_t hwlocObj;
7  hwloc_topology_init(&hwlocObj);
8  HiCR::HWLoc::TopologyManager tm(&hwlocObj);
9  HiCR::HWLoc::MemoryManager mm(&hwlocObj);
10
11 // Obtaining system's CPU topology
12 HiCR::Topology t = tm.queryTopology();
13
14 // Creating Pthread-based Managers
15 HiCR::Pthreads::CommunicationManager cmm;
16 HiCR::Pthreads::ComputeManager cpm;
```

**Figure 4: Example of the start of a HiCR-based application that instantiates a selection of HiCR manager objects. The rest of the application is built upon function calls to these objects.**

Fig. 5 shows an example of instance management operations using the HiCR Core API. This example makes sure the desired number (`N`) of instances have been created at launch time (Line 7). If not, it tries to create however many of them are missing at runtime (Line 12). The new instances, if any, are created based on a template that makes sure they satisfy a given topology. This function is only executed by the root instance (Line 2) to ensure it runs only once.

```
1  // If we are not root, return immediately
2  if (!im.getCurrentInstance().isRoot()) return;
3
4  // Getting launch-time instances count
5  const auto instances = im.getInstances();
6  auto current = instances.size();
7  if (current >= N) return;
8
9  // Creating required instances at runtime
10 auto required = N - current;
11 auto temp = im.createInstanceTemplate(t);
12 im.createInstances(required, temp);
```

**Figure 5: This example make sure the number of instances is at least `N`, either having been initially created by an external launcher, or by creating the number of missing instances at runtime by the instance manager.**

Fig. 6 shows an example where a given message buffer stored in a local memory slot is transferred to all the memory spaces from all devices, as detected by a given topology manager. The communication operations are performed within `for` loops (Lines 2 and 3) that iterate among all the memory spaces within all detected devices and, for each of them, it allocates a new local memory slot (Line 4) and starts a data transfer operation (Line 5). Finally, it makes sure all the communication operations have terminated (Line 7) before returning.

```
1  // Broadcasting message to all local memory spaces
2  for (const auto& d : t.getDevices())
3   for (const auto& s : d.getMemorySpaces()) {
4    auto dst = mm.allocateLocalMemorySlot(s, size);
5    cmm.memcpy(dst, 0, message, 0, messageSize);
6   }
7  cmm.fence();  // Waiting for operations to finish
```

**Figure 6: This example copies a message along all the memory spaces detected by the topology manager. These memory spaces may belong to one or multiple different physical devices on a given node. After initializing the operations, it waits for them to finish before returning.**

Fig. 7 shows an example where a given execution unit (`e`) is simultaneously deployed on a set of compute resources. The code first initializes a processing unit for each of the compute resources provided (Line 5), along with an execution state (Line 6), and starts their execution. The application then waits for all processing units to finish (Line 13), and terminates them. The memory allocated for execution states is freed up as soon as they finish executing.

```
1  // Initializing execution in all compute resources
2  std::vector<HiCR::L0::ProcessingUnit> ps;
3  for (const auto& d : t.getDevices())
4   for (const auto& r : d.getComputeResources()) {
5    auto p = cpm.createProcessingUnit(r);
6    auto s = cpm.createExecutionState(p, e);
7    cpm.initialize(p);
8    cpm.execute(ps, s);
9    ps.push_back(p);
10  }
11
12 // Awaiting finalization
13 for (const auto& p : ps) cpm.await(p);
14 for (const auto& p : ps) cpm.finalize(p);
```

**Figure 7: This example runs an execution unit on all of the available compute resources simultaneously. After initializing their execution, it waits for all processing units to finish executing before returning.**

## 3.2 Built-in Backends

| Backend | Topology | Instance | Communication | Memory | Compute |
|---|---|---|---|---|---|
| MPI | | X | X | X | |
| LPF | | | X | X | |
| YuanRong | | X | | | |
| HWLoc | X | X | | X | |
| Ascend | X | | X | X | X |
| OpenCL | X | | X | X | X |
| Pthreads | | | X | | X |
| Boost | | | | | X |
| nOS-V | | | | | X |

**Table 1: This table lists the built-in backends included with HiCR and the manager classes they implement.**

HiCR can be extended to support any new technologies that satisfy a subset of the core API by developing a new backend plugin for them. Nevertheless, we distribute HiCR with a collection of ready-to-use built-in backends to support several well-established technologies and devices. Table 1 lists the built-in backends currently provided together with the HiCR Core API and the manager classes they implement. An in-depth discussion for each of them follows:

*3.2.1 MPI.* The MPI backend implements operations for communication, memory and instance management using the MPI specification. Its instance manager allows querying how many HiCR instances (i.e., MPI processes) were created at launch time and the unique id for each of them. However, since spawning new processes at runtime is not supported by the majority of MPI implementations, the backend will produce an exception if such an operation is requested. Its memory manager is in charge of allocating new MPI *windows* to serve as source and destination for MPI one-sided communication operations. The communication manager translates HiCR's `memcpy` operations into the corresponding one-sided `MPI_Put` operation, if the source memory slot is local and the destination is global (or the `MPI_Get` operation in the inverse case). Communication between two local memory slots will default to a call to the C `memcpy` operation.

*3.2.2 LPF.* The LPF backend provides support for *Lightweight Parallel Foundations* [4], a communications library following the BSP model [5] for parallel computation. LPF operates mainly by the use of one-sided put and get communication calls whose completion is realized through synchronization calls. Although LPF provides support for multiple *engines* (i.e., communication mechanisms), this backend benefits from LPF's zero-cost synchronization engine which is implemented on top of the Infiniband Verbs API [6, 7] and provides lightweight synchronization mechanisms based on completion queues.

*3.2.3 YuanRong.* *YuanRong* [8] is a serverless platform for general-purpose workloads, and tailored for applications running on cloud infrastructures. The serverless paradigm prescribes that applications should be organized into a set of functions that can be dynamically created based on the incoming workload (e.g., numbers of incoming requests). This backend enables launching a HiCR instance at launch time by wrapping it as a serverless function. The backend assigns a unique id to every new instance at creation time. This procedure allows to emulate the way an SPMD application would work. The root instance is designated to be the one initial instance created by YuanRong at launch time.

*3.2.4 HWLoc.* The HWLoc backend implements topology discovery functionality for CPU-based hosts based on the *Portable Hardware Locality* (hwloc) [9] library. The topology includes a hierarchical view on CPU resources (i.e., sockets, cores, symmetric multithreading) and their memory and cache structures. It also provides a notion of locality, allowing for determining in which NUMA domain each of the compute resources is located. In addition, this backend implements the memory manager class, allowing users to allocate memory slots in either anywhere in the system's RAM, or in a particular NUMA domain.

*3.2.5 Ascend.* This backend enables the use of Ascend AI accelerators [10] in HiCR applications. It offers compute, memory, topology, and communication management capabilities, implemented by the *Ascend Computing Language* [11] platform. The backend's topology manager is in charge of providing a list of all the Ascend accelerators available in the host system. The memory manager enables memory allocation both on accelerators' high memory bandwidth (HBM) memory, as well as on the host memory. For host allocations, the backend guarantees the starting element will be aligned to the start of a pinned page. The use of page-pinning is required for the backend to benefit from direct memory access (DMA) between the device and the host. The communication manager handles the data movement from either the host to the device and vice-versa, or between allocations within the same device, or between two devices. Finally, the compute manager enables the execution of device kernels, building event-based task dependency graphs, and querying their progress.

*3.2.6 OpenCL.* The OpenCL backend enables the discovery and use of both CPU and accelerator devices that support the OpenCL [12] API. This API offers many of the topology, memory, communication, and computation management support for heterogeneous devices, as exposed by the HiCR model. We developed this backend to support the execution of HiCR-based applications outside of [Hidden] 's hardware ecosystem.

*3.2.7 Pthreads.* The Pthreads backend enables threading-based parallelism using the POSIX threads library [13]. Its compute manager enables the creation of processing units, each one of them representing a system-scheduled thread and mapped 1-to-1 to a CPU core or hyperthread, detected as compute resources by the HWLoc backend. The communication manager employs the standard C `memcpy` operation, and guarantees correct completion counts for fencing using POSIX-based mutual exclusion mechanisms.

*3.2.8 Boost.* This backend defines execution units as single (lambda) functions, including the possibility of passing a capture list and a closure argument. It relies on the *Context* library from Boost C++ [14] to instantiate execution units into coroutine-based execution states. These coroutines behave like normal functions, except that they can be suspended and resumed at arbitrary points without the intervention of the OS scheduler. For dependency graph-based runtime systems, these suspension points can be called whenever a task voluntarily yields execution in wait for pending operations.

*3.2.9 nOS-V.* *nOS-V* [15] is a low-level threading and tasking library that enables collaborative co-execution of independent processes. The key feature of nOS-V is its system-wide task scheduler that manages task as a pool of kernel-level threads, all located in a common shared pool across multiple applications. The backend defines a processing unit as a nOS-V task that is bound to execute exclusively on a given CPU core. Similarly, it defines an execution unit as a simple (lambda) function that is instantiated into an execution state represented also by a nOS-V task.

## 3.3 Built-in Frontends

Frontends are ready-to-use libraries that expose higher-level features for communication, execution and distributed computing. The ones presented here are fully based on calls to the HiCR Core API, hence preserving the benefits of an implementation-agnostic approach. Here we discuss their rationale and implementation.

*3.3.1 Channels.* The *Channels* frontend is a communication library tailored for frequent and persistent transfer of small data messages across distributed instances. With this frontend, we target applications that operate on a request basis and typically carry a quality-of-service (QoS) requirement of a low-latency result turnover.

Channels operate by exchanging pre-allocated circular buffers between the sender and receiver instances. These buffers will serve as destination for the transmitted messages. By pre-allocating the buffers, the producer knows where to push the next message, as long as the buffer has not filled up. At that point the producer may not send any more messages until the consumer notifies that a message has been consumed. This logic enables both ends to decouple transfer and synchronization messages, allowing for both minimal per-message handshaking as well as throughput-oriented channel implementations.

The library supports both *Single Producer, Single Consumer* (SPSC) and *Multiple Producer, Single Consumer* (MPSC) paradigms. To support multiple producers, the library offers two operating modes: *locking*, where a collective exclusive access guarantees a shared channel does not overflow, and *non-locking*, where dedicated buffers per producer are employed, eliminating the expensive collective exclusive access, but increasing memory requirements.

*3.3.2 Data Object.* The *Data Object* frontend is a communication library, tailored for sporadic communication of large data objects, such as multi-dimensional tensors. This library allows for performing communication operations without the need of pre-exchanged buffers. Instead, it works by creating a *Data Object* which represents a block of data contained inside a local memory slot, and making it remotely accessible to any other instance by a call to a `publish` operation. Upon publication, the caller obtains a unique data object identifier that can be exchanged with other instances via the Channels frontend.

To access a published data object, other instances perform a `getHandle` operation, which takes a unique identifier as argument and returns a *handle* to the remote object. This handle only represents the required metadata to retrieve the remote object. The data itself can be obtained with a call to `get`, which takes the handle as argument. This function initializes an asynchronous data transfer whose completion can be fenced later using a mechanism similar as described in Figure 6.

*3.3.3 RPC.* The *RPC* frontend offers mechanisms for the registration, listening, and execution of Remote Procedure Calls. This interface is crucial for the initial coordination of execution among multiple instances, especially in the context of applications that rely on instance creation at runtime. RPCs can be used to exchange information about the instance's topology information, establish the creation of communication channels, and to coordinate task execution. To execute an RPC, the function must be pre-registered on the receiving instance. Then, the receiving instance must then enter a listening state either before or after the caller instance launches an RPC request. After execution, the receiving instance may produce a return value that will be automatically returned to the caller.

*3.3.4 Tasking.* The tasking frontend contains building blocks to develop a task-based runtime system. It provides basic support for stateful tasks with settable callbacks to notify when the task changes state, e.g., from *executing* to *finished*. It also contains support for stateful worker objects. These objects contain a simple loop that calls a *pull* function, i.e., a user-defined scheduling function that should return the next task to execute (or a *null* pointer, if none is available). The frontend requires specifying two, possibly distinct, compute managers: one for the worker objects and another for the tasks. In this way, the frontend allows, for example, managing scheduling on the CPU, while executing tasks directly on an accelerator device. This frontend also contains an interface to *OVNI* [16], an instrumentation library to register execution traces. These traces are collected regardless of the computing backend selected and can be loaded into any performance analysis tool.

# 4 Experimental Evaluation

Here we present experimental results based on reproducible test cases programmed directly on the HiCR Core API. The goal of these experiments is to showcase HiCR's adaptability to different execution environments by selecting the appropriate backend in each case. The source code required to reproduce these experiments is available in HiCR's public repository.

## 4.1 Test Case 1: Distributed Ping-Pong

We use a ping-pong pattern to characterize the latency and bandwidth capability of different communication libraries running on a specific interconnect architecture. In contrast to established ping-pong tests such as *NetPIPE* [17], implemented directly on top of low-level system software, this test relies on one-sided operations using HiCR's Channels frontend, for which we showcase portability across the LPF and MPI backends.

This test case involves launching two instances communicating through two SPSC channels for bi-directional communication. It allocates a short message buffer at the consumer side with a fixed single-message capacity. After sending a message (the ping stage), the sender waits on receiving the echoed message (pong stage). This exchange forces a ping-pong pattern, which either results in the latency dominating for small messages, or the goodput (useful data throughput) for large messages.

We designed the benchmark comparing the performance of two backends running on an Infiniband interconnect: (a) LPF backend, relying on IB Verbs-based primitives for Infiniband networks (b) MPI backend, portable across a wide range of networks. We used NVidia's `MLNX_OFED` package version 23.10.2.1.3.1-1, which comes with precompiled Open MPI 4.1.7a1 and corresponding Infiniband drivers. The cluster itself is equipped with Mellanox EDR 100 Gbps Infiniband.

We test with a range of message sizes between $2^{10}$ and $2^{31}$ bytes, repeating each test 10 times and showing vertically the standard deviation. The results of the ping-pong benchmarks are shown in Fig. 8, measuring the *goodput* $G(s)$ across different message sizes $s$ (bytes).

We observe that, for small messages the LPF backend achieves a 70× increase in goodput compared to the MPI-based one. We attribute this to reduced synchronization costs in the LPF backend compared to the MPI backend. The latter displays a less efficient mechanism to keep track of data transfers. For large messages (100s of MBs to few GBs), the speedup almost disappears, and the goodput of both backends converges to 80% of the interconnect's maximum theoretical throughput of 100 Gbps.

Despite the observed results, we emphasize that the MPI-based plugin supports a wider variety of interconnects than the LPF one, which is specifically tailored for Infiniband. Choosing one or the other based on the available interconnect technology is made easy in this example by selecting the appropriate HiCR backend without requiring changes in the code.

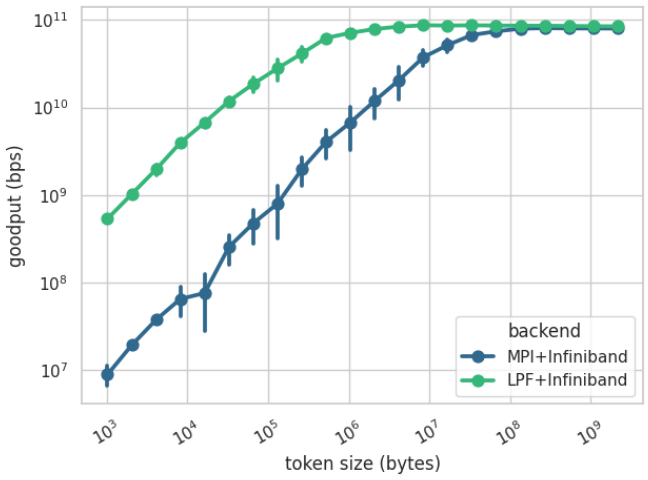


**Figure 8: Observed goodput for ping-pong benchmarks using the LPF (top series) and the MPI (bottom series) backends over multiple message sizes.**

## 4.2 Test Case 2: Heterogeneous Inference

This test case implements a simple HiCR-based forward inference pipeline, typical of AI workloads. For this, we implemented a neural network that takes an encoded image as input and outputs the most likely digit (from 0 to 9) it represents. We used the *MNIST* [18] dataset to train the network and saved its weights for later use during the inference.

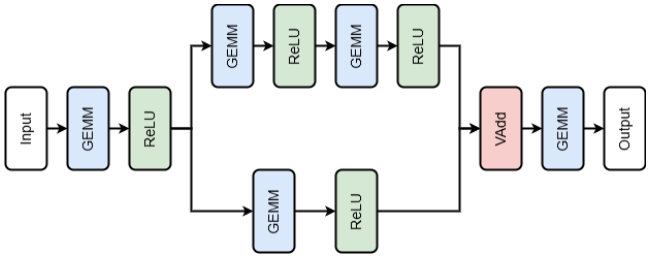


**Figure 9: Neural network architecture used to compute the MNIST dataset. Starting from the input, the image is processed by each layer and when the output is reached thus the prediction is produced. Each GEMM node contains the pre-trained weights.**

The network employs *General Matrix Multiplication* (GEMM), *Vector Add* (VAdd), and *Rectified Linear Unit* (ReLU) kernels, in the order shown in Fig. 9. The application is divided into the following phases: (1) assign the pre-calculated weights to each layer, (2) create HiCR execution units representing the computational kernels, (3) feed the network one image at a time, and (4) output the prediction. While the creation of the execution units is implementation-specific, all the remaining phases are implemented exclusively with calls to the HiCR Core API.

We run the inference on the MNIST test set using three different HiCR backends: Pthreads, Ascend, and OpenCL. For each backend, we provide an appropriate kernel function: Pthreads uses *OpenBLAS* [19] dense linear algebra kernels, Ascend uses pre-compiled kernels provided with the device drivers, and for OpenCL, we manually implemented a naive version of the required kernels.

We run the test case on two computing nodes: (A) containing an Intel Xeon W-1270 CPU and an Intel UHD Graphics P630 GPU, and (B) containing a Huawei Kunpeng 920 CPU and a Huawei Ascend 910A NPU. To prove the correctness of this test, we developed an ad-hoc C++ implementation of the test using OpenBLAS kernels directly without the use of HiCR. We used this as ground truth to verify that the HiCR version produces consistent results on each architecture.

| Device | Node | Backend | Accuracy | img-0 score |
|---|---|---|---|---|
| W-1270 | A | pthreads | 94.64% | 9.921433449 |
| P630 | A | opencl | 94.64% | 9.921431541 |
| Kunpeng 920 | B | pthreads | 94.64% | 9.921433449 |
| Ascend 910A | B | ascend | 94.64% | 9.921875000 |

**Table 2: Inference results. The table reports the percentage of correct predictions over the entire test set (10'000 images), and the highest score computed for img-0 of the test set.**

Table 2 presents the inference results, showing the prediction accuracy for each architecture and backend, and also the exact score given to the most probable digit in the first image of the set (*img-0*). The table shows that, while all backends show consistent accuracies, they still produce slight variations in precision in the outputs. These variations can be attributed to differences in the floating-point operation in the compute kernels and the bit precision between the different devices. Despite these differences in precision,

this test shows that a given compute-intensive HiCR-based application can be executed equally on either CPU or accelerator devices without the need for refactoring.

### 4.3 Test Case 3: Fine-Grained Tasking

This test case calculates a given Fibonacci number $F(n)$ using a naïve approach, i.e., recursively computing $F(n-1)$ and $F(n-2)$ as independent tasks until reaching $F(1)$ and $F(0)$. To schedule the task dependency graph, we employed HiCR's tasking frontend to create a lightweight scheduler that assigns tasks to worker threads as they become available. This test therefore benchmarks the impact of a given backend's compute manager overheads.

We compare two variants containing different configurations for the tasking frontend: (a) uses the nOS-V backend for both worker and tasks management, and (b) Pthreads + Boost for kernel-level thread-based worker instantiation and coroutine-based tasks. For both variants, we compute $F(24)$ with an expected result of 46368, requiring the execution of 150 049 tasks in total. We ran benchmarks on a 44-core dual Intel Xeon Gold 6238T server and report the best measured time among 10 runs. We empirically determined that using 8 worker cores results in the best performance for both variants, and report the results for this configuration only.

We used the OVNI library to obtain the core execution traces of each variant and loaded them into the *Paraver* [20] performance analysis tool for rendering. Fig. 10 shows the traces for the best result for each variant, where nOS-V and Pthreads + Boost backends finish execution in 1.34 and 0.21 seconds, respectively. This example demonstrates that co-routines for quick user-level context switching between fine-grained tasks can greatly reduce overheads compared to instantiating tasks as kernel-level threads while delegating scheduling decisions to the OS scheduler. Applications that do not require such quick context switching may nevertheless prefer the nOS-V variant due to its additional benefits (e.g., thread-level storage or co-location [15]). Regardless of the circumstance, this experiment demonstrates that HiCR can switch between these plugins without modifying application or framework code.

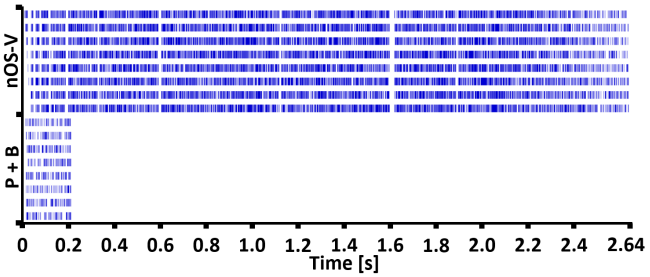


**Figure 10: Execution timelines for the Fibonacci example running on 8 cores. The nOS-V variant (top) finishes in 2.64s, and Pthreads + Boost (P+B, bottom), in 0.21s. Each horizontal line represents the timeline of a distinct CPU core, with solid traces indicating meaningful work and empty spaces representing scheduling overhead.**

### 4.4 Test Case 4: Coarse-Grained Tasking

This test case consists of a parallel three-dimensional iterative heat equation solver that uses the Jacobi method and a 13-point averaging stencil with Manhattan distance one and a finite element grid

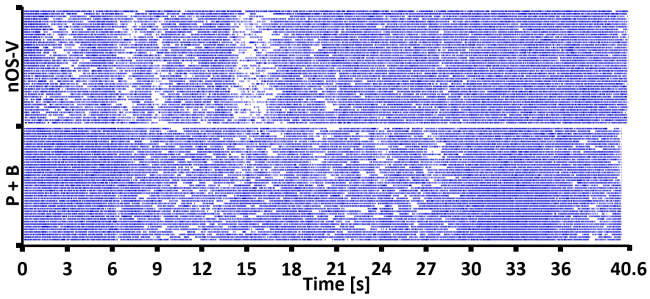


**Figure 11: Execution timelines for the Jacobi example running 500 iterations on 44 Intel Xeon Gold 6238T cores. The nOS-V backend variant (top), finishes in 40.5s, and Pthreads + Boost, in 40.0s.**

of size $704^3$ elements. The application divides the grid across $p^3$ subgrids, each assigned to a different worker thread, and runs a compute-intensive subgrid kernel task whose running time dominates the total runtime of the application. The other tasks in the application encode communication operations between subgrids.

We compare the same variants as the previous test case to run 500 iterations of the solver and report the best measured time among 20 runs. For this case, we use all cores on a 44-core system. Fig. 11 shows the resulting traces for the best run for each variant, where the nOS-V finishes execution in 40.5 s (43.1 GFlop/s), and for Pthreads + Boost, in 39.9 s (43.7 GFlop/s). In this case, we used HiCR to show that the effects of scheduling overheads are minimal when running coarse-grained tasks, regardless of the choice of backend. As a consequence, the additional benefits of nOS-V can be fully exploited without noticeable performance impact.

## 5 Related Work

Both academia and industry have widely adopted the use of models for portable heterogeneous [21] and distributed [22] computing across different hardware and platforms. A large class of solutions centers around the *Partitioned Global Address Space* (PGAS) memory model [23], including programming models such as *Chapel* [24], *X10* [25], and *UPC* [26]. Additionally, Chapel provides abstractions to define kernels to run on accelerators. These solutions work on the level of a memory model, thus directly affecting the application code. Additionally, the use PGAS-based approaches to heterogeneous systems is not straightforward, as it requires the use of ad hoc extensions (e.g., `kernel-launch` in Chapel). In contrast, HiCR provides a direct control over system resources and application scheduling decisions without the need for such extensions.

Metaprogramming models expose annotations (e.g., *pragmas*) in the application code to automatically apply transformations during compilation time for the use of specific technologies. *OpenAcc* [27], *OpenArc* [28] and *HMPP* [29] detect the functions with annotations and make them available to run on accelerator devices while others, such as *OpenMP* [30–32], *XcalableMP* [33, 34], and *OmpSs-2* [35–38] have been extended to support distributed computing as well. Metaprogramming models introduce some complexities in maintaining and extending compiler support for the underlying technologies. Moreover, since platform vendors usually provide their own specialized, often closed-source, compiler implementations, it makes it difficult to extend these implementations

and annotations to new system technologies. In HiCR, all operations are explicit and directly handled by the supporting backends, hence preserving compiler compatibility even when extended to new technologies.

Task-based models organize programs as a collection of tasks and rely on a runtime system to handle scheduling and resource allocation decisions. Tasking models can seamlessly support task execution on heterogeneous devices and across distributed systems by adding specific APIs for such operations. The *Open Community Runtime* (OCR) [39] provides a common layer for building asynchronous many-tasks runtime systems. OCR focuses on runtime-specific building blocks, like tasks and dependencies, and delegates to each particular implementation (e.g., OCR-Vx [40], for OpenCL) the responsibility of dealing with the particulars of the system's hardware and interconnect technology. In contrast to OCR, HiCR does not limit itself to task-based programming as its API is generic enough to describe operations that are portable to any programming model.

Another tasking model, *StarPU* [2], supports heterogeneous and distributed operations via MPI, CUDA, and OpenCL platforms. For each technology, it adopts a corresponding API extension (`starpu_mpi`, `starpu_cuda`, and `starpu_opencl`, respectively) [41]. The drawback in this approach is that either their implementation or API remain fixed to the underlying technologies and are not transferable to any other current or future system. In contrast, HiCR plugins ensure that applications can seamlessly be adapted to future technologies.

IRIS [42] is an implementation-agnostic runtime system with heterogeneous hardware support. Similar to HiCR, it hides the device characteristics behind a unique abstract model capable of representing a wide range of hardware. However, it does not offer support for distributed computation and prescribes a programming model that requires an application to be conceived in terms of task dependencies.

Other run-time system approaches like *HPX* [43], *SkePU* [44], *XKaapi* [45], *Minos Computing Library* [46], *Specx* [47], *Charm++* heterogeneous extension [48], and *DuctTeip* [49] share the similarities and differences with HiCR of the aforementioned approaches. On the other hand, *oneAPI* [50], *HIP* [51], *Celerity* [52], *OCAL* [53], *C++ AMP* [54], *SYCL* [55], *Nvidia Thrust* [56] allow for general-purpose programming over multiple devices within a single node, but do not handle the aspect of distributed computing.

Finally, domain-specific solutions, such as *Kokkos* [57], *RAJA* [58], *AllScale* [59], and *Data Parallel C++* [60] offer language extensions or APIs for mathematical, physical, or related field operations, targeting parallel and heterogeneous computing systems. In contrast to these, HiCR's approach requires no language extensions or domain-specific interfaces.

Tools like COMP Superscalar [61] enable deployment on cloud infrastructures and dynamic provision of resources in a transparent way to end-users, relieving them from refactoring applications to adapt to different cloud providers. Such functionality is in line with what HiCR ascribes to the instance manager. HiCR, however integrates this functionality into a complete model that includes memory, compute and communication operations.

## 6 Conclusions and Future Work

Distributed heterogeneous systems present new complexities to modern HPC and AI technology. Developing applications in these fields in a way that they can adapt to the latest and upcoming hardware can be a daunting endeavor. We have presented a model to minimize such an effort, enabling programmers to express the semantics of applications without dealing with the underlying implementation complexities.

Contrary to other solutions, HiCR describes a set of abstract operations without prescribing any particular programming model or implementation decisions. As a result, many of the aforementioned projects could, in fact, be constructed on top of HiCR Core API operations. Finally, we have shown empirically that applications programmed with HiCR can run on multiple architectures or with different supporting libraries while preserving its semantics.

We have deployed HiCR as the supporting library for an internal [Hidden] distributed runtime system. The runtime system employs HiCR's built-in frontends to support the execution of two applications. The first represents a online service for multimodal inference, using YuanRong backend for cloud-based instance management. The second application runs an electronic design automation (EDA) solver on a datacenter setting, using the MPI backend for the initial instance deployment and communication.

Future work includes extending the model for discovery of the interconnect topology. Most communication libraries assume fully homogeneous connectivity, essentially assuming the BSP computer model [5], relying on the interconnect to most favourably resolve routing. However, having a complete picture of the topology may enable more intelligent algorithms and collectives communication operations. HiCR could also include cost models for its operations. For instance, it might be useful to associate a latency and bandwidth capabilities to both memory spaces (e.g., in NUMA systems) and interconnect links [4]. Finally, (distributed) file management, multi-user job allocation, and security isolation could be included in the model.

## References


[1] MPI Forum. https://www.mpi-forum.org/. (2025-03-24).

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par Parallel Processing (Lecture Notes in Computer Science, Vol. 5704)*. Springer, 863–874.

[3] Nvidia CUDA. https://developer.nvidia.com/cuda-toolkit. (2025-03-24).

[4] Wijnand Suijlen and A. N. Yzelman. 2019. Lightweight Parallel Foundations: a model-compliant communication layer. *CoRR* abs/1906.03196 (2019).

[5] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[6] Introduction to Infiniband. https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf. (2025-03-24).

[7] Infiniband Verbs API. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/vpi+verbs+api. (2025-03-24).

[8] Qiong Chen et al. 2024. YuanRong: A Production General-purpose Serverless System for Distributed Applications in the Cloud. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 843–859.

[9] Portable Hardware Locality (hwloc). https://www.open-mpi.org/projects/hwloc. (2025-03-24).

[10] Huawei Ascend Computing. https://e.huawei.com/en/products/computing/ascend. (2025-03-24).

[11] Huawei Ascend Computing Language. https://www.hiascend.com/document/detail/zh/canncommercial/700/overview/index.html. (2025-03-24).

[12] Open Computing Language (OpenCL). https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html. (2025-03-24).

[13] POSIX Threads. https://man7.org/linux/man-pages/man7/pthreads.7.html. (2025-03-24).
[14] Boost Context. https://www.boost.org/doc/libs/1_84_0/libs/context/doc/html/index.html. (2025-03-24).
[15] David Álvarez, Kevin Sala, and Vicenç Beltran. 2024. nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 312–324.
[16] Obtuse but Versatile Nanoscale Instrumentation (ovni). https://ovni.readthedocs.io/. (2025-03-19).
[17] Quinn O Snell, Armin R Mikler, and John L Gustafson. 1996. Netpipe: A network protocol independent performance evaluator. In *IASTED international conference on intelligent information management and systems*, Vol. 6. Washington, DC, USA), 49.
[18] MNIST Dataset. https://www.kaggle.com/datasets/hojjatk/mnist-dataset. (2025-03-24).
[19] OpenBLAS: An optimized BLAS library. https://www.openmathlib.org/OpenBLAS/. (2025-03-24).
[20] Paraver: a flexible performance analysis tool. https://tools.bsc.es/paraver. (2025-03-19).
[21] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4 (2015), 69:1–69:35.
[22] Thomas L. Sterling, Matthew Anderson, and Maciej Brodowicz. 2017. A Survey: Runtime Software Systems for High Performance Computing. *Supercomput. Front. Innov.* 4, 1 (2017), 48–68.
[23] 2011. Partitioned Global Address Space (PGAS) Languages. In *Encyclopedia of Parallel Computing*, . Springer, 1465.
[24] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. 2004. The Cascade High Productivity Language. In *International Workshop on High-Level Programming Models and Supportive Environments*. IEEE Computer Society, 52–60.
[25] Philippe Charles et al. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 519–538.
[26] Tarek A. El-Ghazawi and Lauren Smith. 2006. UPC - UPC: unified parallel C. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing*. ACM Press, 27.
[27] Open Accelerators (OpenACC). https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf. (2025-03-24).
[28] Seyong Lee and Jeffrey S. Vetter. 2014. OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *The International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 115–120.
[29] Romain Dolbeau, Stéphane Bihan, and François Bodin. 2007. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units*, Vol. 28. Citeseer.
[30] Open Multi-Processing (OpenMP). https://www.openmp.org/specifications/. (2025-03-24).
[31] Pedro Valero-Lara, Jungwon Kim, Oscar R. Hernandez, and Jeffrey S. Vetter. 2021. OpenMP Target Task: Tasking and Target Offloading on Heterogeneous Systems. In *Euro-Par: Parallel Processing Workshops (Lecture Notes in Computer Science, Vol. 13098)*. Springer, 445–455.
[32] OmpCluster. https://ompcluster.gitlab.io/. (2025-03-24).
[33] XcalableMP. https://xcalablemp.org/index.html. (2025-03-24).
[34] XcalableACC. https://xcalablemp.org/XACC.html. (2025-03-24).
[35] OmpSs-2. https://pm.bsc.es/ompss-2. (2025-03-24).
[36] Jimmy Aguilar Mena et al. 2022. OmpSs-2@ Cluster: Distributed memory execution of nested OpenMP-style tasks. In *European Conference on Parallel Processing*. Springer, 319–334.
[37] Eduard Ayguadé et al. 2009. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Euro-Par Parallel Processing (Lecture Notes in Computer Science, Vol. 5704)*. Springer, 851–862.
[38] Vinoth Krishnan Elangovan, Rosa M. Badia, and Eduard Ayguadé Parra. 2012. OmpSs-OpenCL Programming Model for Heterogeneous Systems. In *Languages and Compilers for Parallel Computing (Lecture Notes in Computer Science, Vol. 7760)*. Springer, 96–111.
[39] Timothy G. Mattson et al. 2016. The Open Community Runtime: A runtime system for extreme scale computing. In *IEEE High Performance Extreme Computing Conference*. IEEE, 1–7.
[40] Jirí Dokulil and Siegfried Benkner. 2022. The OCR-Vx experience: lessons learned from designing and implementing a task-based runtime system. *J. Supercomput.* 78, 10 (2022), 12344–12379.
[41] Cédric Augonnet et al. 2012. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In *Recent Advances in the Message Passing Interface - European MPI Users' Group Meeting (Lecture Notes in Computer Science, Vol. 7490)*. Springer, 298–299.
[42] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2021. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In *IEEE High Performance Extreme Computing Conference*. IEEE, 1–8.
[43] Hartmut Kaiser et al. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the International Conference on Partitioned Global Address Space Programming Models*. ACM, 6:1–6:11.
[44] Johan Enmyren and Christoph W Kessler. 2010. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*. 5–14.
[45] Thierry Gautier, João V. F. Lima, Nicolas Maillard, and Bruno Raffin. 2013. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *IEEE International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1299–1308.
[46] Roberto Gioiosa et al. 2020. The Minos Computing Library: efficient parallel programming for extremely heterogeneous systems. In *Annual Workshop on General Purpose Processing using Graphics Processing Unit*. ACM, 1–10.
[47] Paul Cardosi and Bérenger Bramas. 2023. Specx: a C++ task-based runtime system for heterogeneous distributed architectures. *CoRR* abs/2308.15964 (2023).
[48] Michael P. Robson, Ronak Buch, and Laxmikant V. Kalé. 2016. Runtime Coordinated Heterogeneous Tasks in Charm++. In *International Workshop on Extreme Scale Programming Models and Middleware*. IEEE Computer Society, 40–43.
[49] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. 2019. DuctTeip: An efficient programming model for distributed task-based parallel computing. *Parallel Comput.* 90 (2019).
[50] Robert W. Wisniewski et al. 2021. A Holistic Systems Approach to Leveraging Heterogeneity. In *IEEE/ACM Programming Environments for Heterogeneous Computing*. IEEE, 27–33.
[51] AMD Heterogeneous-computing Interface for Portability (HIP). https://rocm.docs.amd.com/projects/HIP/en/latest/. (2025-03-24).
[52] Peter Thoman, Philip Salzmann, Biagio Cosenza, and Thomas Fahringer. 2019. Celerity: High-Level C++ for Accelerator Clusters. In *Euro-Par: Parallel Processing (Lecture Notes in Computer Science, Vol. 11725)*. Springer, 291–303.
[53] Ari Rasch, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. 2018. OCAL: An Abstraction for Host-Code Programming with OpenCL and CUDA. In *IEEE International Conference on Parallel and Distributed Systems*. IEEE, 408–416.
[54] C++ Accelerated Massive Parallelism (C++ AMP). https://learn.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-overview?view=msvc-170. (2025-03-24).
[55] SYCL. https://www.khronos.org/sycl/. (2025-03-24).
[56] Nvidia Thrust. https://developer.nvidia.com/thrust. (2025-03-24).
[57] H. Carter Edwards and Daniel Sunderland. 2012. Kokkos Array performance-portable manycore programming model. In *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 1–10.
[58] David Beckingsale et al. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC*. IEEE, 71–81.
[59] Herbert Jordan et al. 2020. The allscale framework architecture. *Parallel Comput.* 99 (2020), 102648.
[60] Data Parallel C++. https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html. (2025-03-24).
[61] Enric Tejedor and Rosa M. Badia. 2008. COMP Superscalar: Bringing GRID Superscalar and GCM Together. In *IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 185–193.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009