HiCR, an Abstract Model for Distributed Heterogeneous Programming

Sergio Miguel Martin
Luca Terracciano
Kiril Dichev
Noah Baumann
Orestis Korakitis*
Computing Systems Laboratory
Huawei Zurich Research Center
Switzerland

Jiashu Lin
HiSilicon Technologies
China

Albert-Jan Yzelman†
Computing Systems Laboratory
Huawei Zurich Research Center
Switzerland
albertjan.yzelman@huawei.com

Abstract

We present HiCR, a model to represent the semantics of distributed heterogeneous applications 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 systems without the need for significant refactoring, while also being able to serve any governing parallel programming paradigm. In terms of software abstraction, HiCR is naturally located between distributed heterogeneous systems and runtime systems. We coin the phrase *Runtime Support Layer* for this level of abstraction. We explain how the model's components and operations are realized by a plugin-based approach that takes care of device-specific implementation details, and present examples of HiCR-based applications that operate equally on a diversity of platforms.

1 Introduction

Recent advancements in artificial intelligence models present substantial demands to modern computing systems, as they 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 or data centers.

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 and runtime systems. The model exposes a minimal set of 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 code will reach its intended result regardless of the system it executes on.

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, code written using HiCR need only be written once to seamlessly execute across a wide range of technologies and platforms. Second, any newly developed plugin automatically becomes available to all HiCR-based code, allowing them to benefit from the added functionality without further modifications.

The contributions of this paper are: (a) the introduction of a *Runtime Support Library* as an intermediate layer between an application and its programming framework or runtime system, and its underlying technologies; (b) the proposal of HiCR, as an abstract model for the operations that this layer should offer; (c) an open-source implementation of the model, and; (d) its functional verification through experiments.

*Affiliation at the time of contribution.
†Corresponding author.

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

2 Related Work

This Section discusses similarities and differences of HiCR with, in turn: runtime systems based on metaprogramming (e.g., *pragmas*), tasking runtime systems, skeleton and automatic programming frameworks, as well as (cloud) workflow middleware.

Metaprogramming frameworks expose annotations in the application code to automatically apply transformations during compilation time for the use of specific technologies. *OpenACC* [46], *OpenArc* [35] and *HMPP* [21] detect the functions with annotations and make them available to run on accelerator devices while others, such as *OpenMP* [44, 48, 61], *XcalableMP* [63, 64], and *OmpSs-2* [2, 7, 24, 45] have been extended to support heterogeneous, and in some cases, distributed-memory parallel computing. These programming frameworks provide an advantage when adapting existing code to parallel execution, whereas using HiCR for this purpose may require significant refactoring. However, metaprogramming also introduce complexities in maintaining and extending compiler support for the underlying technologies. Moreover, platform vendors usually provide their own specialized, often closed-source, compiler implementations, thus hindering the extension of these annotations to new system technologies. With HiCR, all operations are explicit and directly handled by the supporting backends, hence preserving compiler compatibility even when extended to new technologies. It also allows for proprietary backends without sacrificing portability.

Task-based programming frameworks organize programs as a collection of tasks and rely on a runtime system to make scheduling and resource allocation decisions. They seamlessly support task execution on heterogeneous devices and across distributed systems by adding specific APIs for such operations. For example, the *StarPU* [6] runtime system interfaces directly with MPI and OpenCL or *CUDA* [41] for distributed communication and GPU operations, respectively. For each technology, it adopts corresponding API extensions (`starpu_mpi`, `starpu_cuda`, and `starpu_opencl`) [5]. The drawback is that either their implementation or API remain fixed to the underlying technologies and are not transferable to any other current or future system. Instead, the HiCR model ensures that applications can seamlessly adapt to future technologies by developing corresponding backends.

Efforts from academia and industry have sought automatic frameworks for distributed and heterogeneous computing that allow expressing portable high-level code. One approach revolves around *skeletons*, mini-programs with prescribed input and output structures that can automatically parallelize and dispatch to accelerators and whose composition describes higher-order functionalities. These include *SkePU* [25], *Kokkos* [22], *RAJA* [8], *AllScale* [32], *SYCL* [58], and *Data Parallel C++* [17]. In contrast to these approaches, HiCR requires no language extensions nor does it prescribe specific skeletons. PGAS [1] inspired another set of approaches, leading to frameworks such as *Chapel* [13], *X10* [15], and *UPC* [23] – while a final class of frameworks center around BSP and its automatic PRAM simulation [62] which led to frameworks such as MapReduce [18], Pregel [37], and Spark [67]. Applying these approaches to heterogeneous systems is not straightforward; for example, Chapel provides abstractions to define kernels that dispatch to accelerators via ad-hoc extensions such as `kernel-launch`, and similar for other automatic frameworks [65]. In contrast, HiCR provides direct control over system resources and application scheduling decisions without the need for such extensions.

Distributed memory frameworks such as *COMP Superscalar* [59] 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 its *instance manager*. HiCR, however, integrates this functionality into a complete model that also includes memory, compute and communication operations.

Other runtime frameworks [11, 14, 26, 27, 33, 54, 66] share the similarities and differences with HiCR of the aforementioned approaches. Others [4, 12, 42, 53, 60] allow for general-purpose programming over multiple devices within a single node, but do not handle the aspect of distributed computing. Of these, IRIS [34] supports heterogeneous hardware while, similar to HiCR, hiding the device characteristics behind a unique abstract model capable of representing a wide range of hardware. Aside from not supporting for distributed computation, unlike HiCR, IRIS prescribes task-based programming.

The *Open Community Runtime* (OCR) [38] 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 [19] for OpenCL) the responsibility for dealing with particular hardware and interconnect technologies. These aims are similar to that of a Runtime Support Layer. However, OCR performs no resource provisioning while implementations must take on dependency graph management and data block management. For distributed-memory implementations the latter are both challenging and prescribe, to significant degree, the design of any task-based programming framework based on it [20]. By contrast, the

HiCR API is generic enough to support any programming model, tasking or otherwise, while retaining full freedom as to their implementation.

3 The HiCR Model

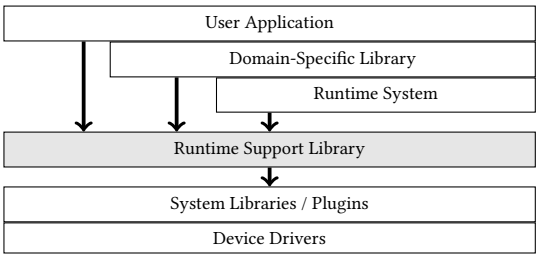

Figure 1. A layered view of runtime support for user applications. Any of the layers may invoke functionality of one or more of the layers below it to support runtime operations.

Given the complexity of modern computing systems, applications typically rely on a multitude of third-party and system libraries. 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, it may suffice to only require access to device drivers (low-level interrupts) or system libraries (e.g., *MPI* [40]). 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 conducting scheduling 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. This approach results in limited portability to other existing or future technologies if the underlying runtime system is not updated accordingly. However, even if it is, updates typically imply runtime system API changes which, in turn, requires refactoring at the application level.

To solve the issue of portability, we propose the concept of a *Runtime Support Layer* to serve as a bridge from application and runtime systems to underlying low-level system libraries. This layer provides an implementation-agnostic API consisting of computation, communication, and system management building blocks. We propose HiCR as a model to describe such building blocks.

HiCR comprises a minimal set of components and operations to describe the semantics of any code running on any distributed computing system. That is, it assumes the existence of interconnected computing nodes, each comprised of processing units connected to local memory. In addition, the model imposes no semantic prescriptions to the way an application must be programmed or executed. Instead, it can be employed equally well by a programming framework such as a tasking runtime system as well as directly by an application, or indeed by any middleware layer in-between. By decoupling the runtime support library from the programming model, we thus enable HiCR to be used by highly diverse programming approaches: any runtime system, programming framework, 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 *backends*, of the HiCR model. In this way, a wide variety application may be ported to other systems and technologies simply by selecting different sets of backends.

3.1 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 computation operations, the copying of data from one device to another, or create a new application instance. In addition, only managers can create instances of other components, both stateless and stateful.

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.

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.

3.1.1 Instance Management. The model defines an *Instance* as any subset of the entire distributed 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. Full access is typical for bare-metal deployments, while partial access is typical for virtualized, e.g., cloud-based deployments. 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) initiates 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, which is typical

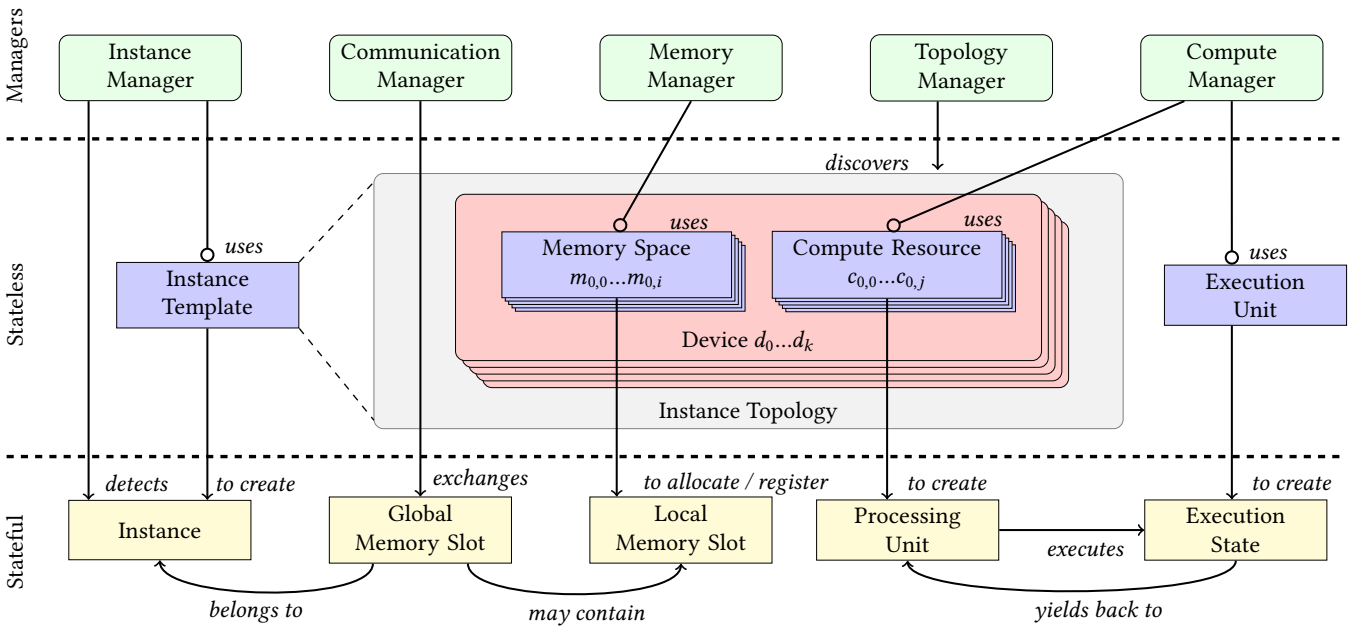

Figure 2. Diagram showing the components of the HiCR model and the available operations between them. The model is divided in three component groups: *Managers*, components whose operations represent an application's semantic building blocks; *Stateless*, components that represent static information,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

for applications that deploy on cloud infrastructures. In this case, the instance manager, running on the initial instance, requests (e.g., to a cloud-service provider) the ramp up of new hosts. Creating a new instance requires passing an *Instance Template* object to the instance manager. This object encapsulates the description of a required topology plus any custom metadata accepted by the underlying technology. This template prescribes the minimal hardware resources required from 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.

3.1.2 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., a NUMA Domain or a GPU), containing zero or more memory spaces and compute resources.

A *Memory Space* represents a hardware element that exposes explicitly addressable 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 accelerator devices, memory spaces may include device RAM, addressable caches, as well as high-bandwidth memories. Future or non-standard memories, such as large capacity or sequential access, could be represented as long as they fit the definition here provided.

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 the full information of all the devices comprising the local instance. This information can be serialized and broadcast, allowing users to build a topological picture of the entire distributed system.

3.1.3 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 `std::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 (and thus, the device) 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 manual registration of an existing memory allocation as a new local memory slot 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 existing allocation received externally (e.g., from a math library).

3.1.4 Communication Management. All communication in the model is mediated by the *Communication Manager* via its `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*, local memory slots that are made accessible to other HiCR instances and can be used as the source or destination of distributed `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 distinguishes the resulting global memory slots.

Only three directions are allowed for the `memcpy` operation: *Local-to-Local*, *Local-to-Global*, and *Global-to-Local*. The first entails 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`, respectively) may be used. 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 communication between two remote 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. If available, the communication manager will use the accelerator-specific network to satisfy the request, instead of channeling the transfer through the host network.

3.1.5 Compute Management. The *Compute Manager* is in charge of carrying out 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 (i.e., a compute resource) may be initialized as a POSIX thread with a one-to-one binding to that core. Similarly, a processing unit may represent a stream context in an accelerator device. 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 core, or; pre-compiled kernels, for 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 user asks a compute manager to create a new execution state, derived from a given execution unit. The user 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 is considered finished and cannot be re-used.

4 Implementation

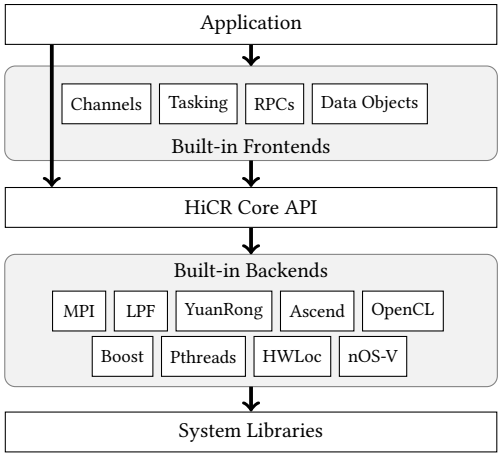

Figure 3. The current implementation of the HiCR model. Its components and operations are exposed in a *Core API*, which serves as interface between the user-level applications and the underlying system libraries. The core API is distributed together with a set of built-in *backends*, plugins containing 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 library[1]. The library is distributed together with a set of built-in backend modules. *Backends* are ready-to-use plugins 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 abstract API, allowing a HiCR-based program to preserve its semantics across a diversity of devices. Fig. 3 shows how these components serve to relate the user application code to the underlying system libraries.

Built-in backends and their system library dependencies can be included in the compilation chain by configuring them in HiCR's *meson*-based build system during setup time. Any future or third-party backends can be supported by manually adding the corresponding compilation flags. Built-in backends are described in Section 4.2; we first discuss the HiCR interface.

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. An up-to-date list of existing frontends can be found in HiCR's

[1] https://github.com/Algebraic-Programming/HiCR

4.1 Programming with HiCR

All the components of the HiCR model are implemented as C++ abstract classes. As a result, they cannot be instantiated directly. Instead, each of the backends derives them into complete classes by implementing their pure virtual functions. The user is therefore required to first instantiate the appropriate backends and then pass them to a HiCR-based application as inputs.

Fig. 4 shows an example of backend instantiation, prior to running a HiCR application. 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. Finally, it instantiates the *Pthread*-based communication and compute managers (Lines 12 and 13). These managers are passed by reference or pointer to a HiCR application, which receives them as abstract classes and thus remains agnostic to the specific choice of backends.

```
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 // Creating Pthread-based Managers
12 HiCR::Pthreads::CommunicationManager cmm;
13 HiCR::Pthreads::ComputeManager cpm;
```

Figure 4. Example of backend instantiation. The resulting manager objects are passed to a HiCR application which is built exclusively with calls to abstract HiCR classes.

4.1.1 Example: Inter-Device Communication.

Fig. 5 shows an example where a given message buffer, stored in a local memory slot, is transferred to all memory spaces at all devices. Since this and the following examples represent pure HiCR applications, all manager objects are pointers to abstract HiCR classes, which are agnostic to implementation decisions. For instance, the code obtains the system's hardware topology from `tm`, a pointer to the abstract `HiCR::TopologyManager` class and stores it in a HiCR `Topology` object (Line 2). The communication operations are performed within `for` loops (Lines 5 and 6) that iterate among all the memory spaces within all detected devices in the topology and, for each of them, it allocates a new local memory slot (Line 7) and starts a data transfer operation (Line 8). Finally, it makes sure all the communication operations have terminated (Line 10) before returning.

```
1  // Obtain system's CPU topology
2  HiCR::Topology t = tm->queryTopology();
3
4  // Broadcast message to all local memory spaces
5  for (const auto& d : t.getDevices())
6   for (const auto& s : d.getMemorySpaces()) {
7    auto dst = mm->allocateLocalMemorySlot(s, size);
8    cmm->memcpy(dst, 0, message, 0, messageSize);
9   }
10 cmm->fence();  // Wait for operations to finish
```

Figure 5. 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.

```
1  // Initializing execution in all compute resources
2  std::vector<HiCR::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(p, 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 6. This example runs a given execution unit on all of the available compute resources for parallel execution.

4.1.2 Example: Parallel Execution.

Fig. 6 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 then starts their execution (Lines 7 and 8). The application then waits for all processing units to finish and terminates the execution states (Lines 13 and 14), freeing the memory allocated for them.

4.1.3 Example: Distributed Deployment.

Fig. 7 shows an example of instance management operations using the HiCR Core API. This example checks whether the `desired` number of instances have been created at launch time (Line 9). If not, it tries to create, at runtime, however many of them are found missing (Line 14). The new instances, if any, are created based on a template that makes sure they satisfy a given set of requirements (`reqs`), which may include, but are not limited to, hardware or network topology specifications. This snippet is only executed by the root instance (Line 2) to ensure that it runs exactly once.

4.2 Built-in Backends

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. Additionally, we distribute with HiCR a collection of ready-to-use built-in backends to support several well-established technologies and devices. Table 1 shows the

```
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
8  // Return if the desired instances already exist
9  if (current >= desired) return;
10
11 // Creating required instances at runtime
12 auto required = desired - current;
13 auto temp = im->createInstanceTemplate(reqs);
14 im->createInstances(required, temp);
```

Figure 7. This example makes sure there are `desired` number of instances by either having been initially created by an external launcher, or by creating new instances at runtime.

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

Table 1. This table lists the backends currently provided and the subset of the HiCR model that they implement.

current list of the built-in backends and the manager classes they implement. We chose to provide these backends as they represent a mixture between commonly used technologies with more research-oriented ones.

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. The backend instantiates memory slots as MPI *windows* to serve as source and destination for MPI one-sided communication operations and translates distributed HiCR `memcpy` operations into the corresponding one-sided `MPI_Put` or `MPI_Get` operations.

The *LPF* backend provides support for *Lightweight Parallel Foundations* [57], a communications library following the BSP model [62] for parallel computation. LPF operates mainly by the use of one-sided put and get communication calls whose completion is realized through synchronization calls. This backend uses a synchronization engine implemented on top of the Infiniband Verbs API [30, 31] and provides lightweight synchronization mechanisms based on completion queues.

YuanRong [16] 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 as a serverless function at runtime.

The *HWLoc* backend implements topology discovery functionality for CPU-based hosts based on the *Portable Hardware Locality* [51] 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 and allocating memory on any such domains.

The *ACL* backend provides support for Huawei NPU accelerators [28, 29], exposing a list of such devices available in the host system and enabling memory allocation both on accelerators' high memory bandwidth (HBM) memory, as well as on the host memory. It allows data motion from either the host to the device and vice-versa, between allocations within the same device, or; between two devices. It also enables the execution of device kernels, building event-based dependency graphs, and querying their progress.

The *OpenCL* backend enables the discovery and use of both CPU and accelerator devices that support the OpenCL [47] API. This API offers many of the topology, memory, communication, and computation management support for heterogeneous devices.

The *Pthreads* backend enables threading-based parallelism using the *POSIX Threads* library [52]. 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 fencing using mutual exclusion mechanisms.

The *Boost* 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++ [10] 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.

The *nOS-V* backend enables the use of nOS-V [3], a low-level threading library that enables collaborative co-execution of independent processes. nOS-V features a system-wide scheduler that assigns each task to its own kernel-level thread, all located in a common shared pool across multiple processes.

4.3 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.

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.

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 5.

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.

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* [43], 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.

5 Experimental Evaluation

Here we present experimental results based on reproducible test cases programmed exclusively with calls to the HiCR API. The goal of these experiments is to demonstrate the adaptability of HiCR-based applications to different execution environments by selecting the appropriate backend for each case, without requiring changes in its source code. The source code required to reproduce these experiments is available in HiCR's public repository.

5.1 Test Case 1: Communication Benchmark

This test case involves launching two instances communicating through two opposing single-producer single-consumer 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 (ping), the sender waits on receiving the echoed message (pong). This exchange forces a ping-pong pattern (similar to a one-sided version of the two-sided *NetPIPE* [56]), which results in either latency-bound results when communicating small messages, or throughput-bound results for large messages.

We designed the benchmark to compare the performance of two backends: (a) the LPF backend, relying on its *zero* engine [36] that employs primitives for Infiniband networks, and (b) the MPI backend. All tests were conducted on a cluster equipped with a Mellanox EDR 100 Gbps Infiniband fabric. We run the test with a range of message sizes from 1 byte to ≈ 2.14 Gigabytes, repeating each test 10 times and showing the standard deviation vertically. The results of the ping-pong benchmarks are shown in Fig. 8, measuring the *goodput* $G(s)$ (i.e., effective throughput) across different message sizes.

For small messages, the LPF backend achieves a 70× increase in goodput compared to the MPI-based one. We attribute this to LPF making direct use of hardware-enabled Infiniband completion queues that minimizes handshaking. This extends beyond what the MPI standard currently allows, similarly to other proposed optimizations [9, 55]. The MPI backend employs standard one-sided MPI primitives that induce less efficient handshaking. The larger messages ($> 10^9$ bytes) are far less affected by handshaking latencies, and therefore indeed the goodput of both backends converges to ~80% of the maximum theoretical throughput of the interconnect: 100 Gbps.

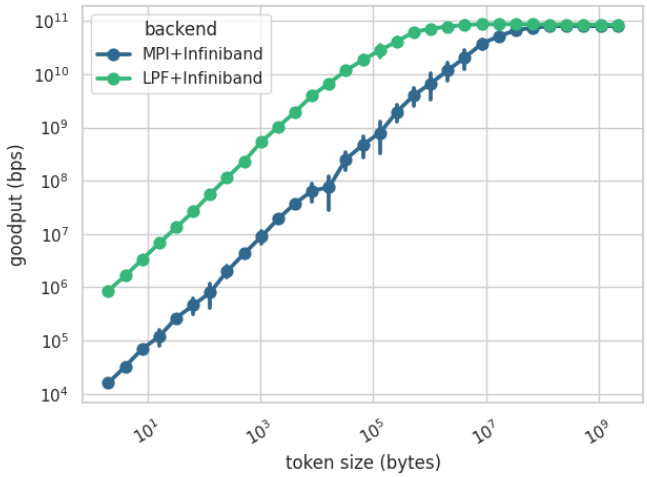

Figure 8. Observed goodput for ping-pong benchmarks using the LPF (top series) and MPI (bottom series) backends over multiple message sizes. LPF relies on IBverbs directly while MPI relies on OpenMPI RMA primitives.

5.2 Test Case 2: Heterogeneous Inference

This test case implements a simple forward inference pipeline, mimicking typical AI inference workloads. We implemented a HiCR application featuring a neural network that takes an encoded image as input and outputs the most likely digit (from 0 to 9) that they represent. We used the *MNIST* [39] dataset to train the network and saved its weights for later use during the inference. We run the application passing to it three different backends: Pthreads, ACL, and OpenCL. For each backend, we provide an appropriate kernel function: the Pthreads variant runs *OpenBLAS* [49] dense linear algebra kernels; the ACL variant uses pre-compiled kernels provided with the NPU device, and; the OpenCL variant uses a naïve version of the required kernels. We conducted experiments on two computing nodes: (A) containing an Intel Xeon W-1270 CPU and an Intel P630 GPU, and (B) containing a Huawei Kunpeng 920 CPU and a Huawei 910A NPU. To prove the correctness, we developed an ad-hoc C++ implementation of the test using OpenBLAS kernels directly

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
Huawei 910A	B	acl	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.

without the use of HiCR. We used the latter to verify that all variants produce consistent results on each architecture.

Table 2 presents the inference results, showing the prediction accuracy for each architecture and backend, and also the highest 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. These variations can be attributed to differences in the floating-point precision of the devices and to differing orders of computations in the compute kernels. Despite these, this test shows that a given HiCR-based application can be executed equally on either CPU or accelerator devices by providing the appropriate backend and the kernels to run its operations.

5.3 Test Case 3: Fine-Grained Tasking

This test case computes the 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 developed a HiCR-based lightweight scheduler[2] that assigns tasks to worker threads as they become available. Since we use the exact same setup for all runs, this test is designed to compare the compute performance between different HiCR backends in a way that measures the overheads of context switching.

We employ two HiCR variants: (a) one using the nOS-V threading backend for both worker and tasks management, and; (b) another using Pthreads + Boost for thread-based workers and coroutine-based tasks, respectively. For both variants, we compute $F(24)$ with an expected result of 46 368, requiring the execution of 150 049 tasks in total. We ran benchmarks on a dual-socket 22-core Intel Xeon Gold 6238T server with hyperthreading enabled, and report the best measured time among 10 runs. We determined that using 8 worker threads that are pinned to individual cores in the same socket yields the best performance for both variants.

We used the OVNI library to obtain the core execution traces and visualize them with *Paraver* [50]. Fig. 9 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 user-level context switching between fine-grained tasks can greatly reduce overheads compared to delegating scheduling decisions to the OS.

[2]https://github.com/Algebraic-Programming/TaskR

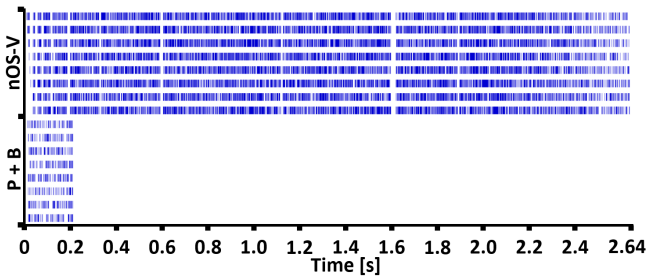

Figure 9. Execution timelines for the Fibonacci example that executes 150 049 tasks using 8 cores. Each horizontal line represents the timeline of a CPU core, with solid traces indicating meaningful work and empty spaces indicating scheduling overhead.

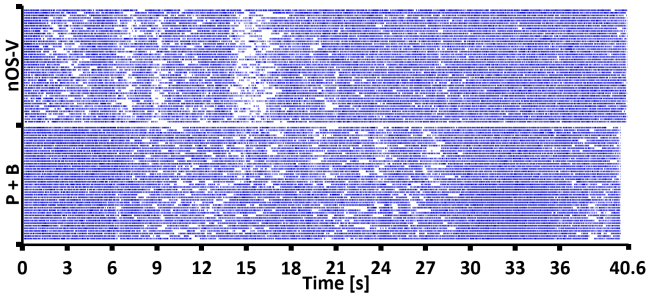

Figure 10. Execution timelines for the Jacobi example running 500 iterations using $1 \times 2 \times 22 = 44$ threads of a Intel Xeon Gold 6238T system with hyperthreading enabled (but here unused).

5.4 Test Case 4: Coarse-Grained Tasking

Consider a 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 of size 704^3 elements. Our test case divides the grid, consisting of a single contiguous allocation, across $l_x \times l_y \times l_z$ local subgrids, and assigns each to a unique worker thread. Each thread thus executes an expensive local computation that dominates the total runtime, followed by communication of the subgrid halos. This process repeats for a set number of iterations.

We compare the nOS-V and Pthreads+Boost variants over 500 iterations of the solver and report the best measured time among 20 runs. For this case, we employ the same dual-socket system as in Section 5.3 and achieve best performance using a $1 \times 2 \times 22$ thread grid, thus spawning a total of 44 threads that are each assigned a core; i.e., not using hyperthreading. Fig. 10 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.

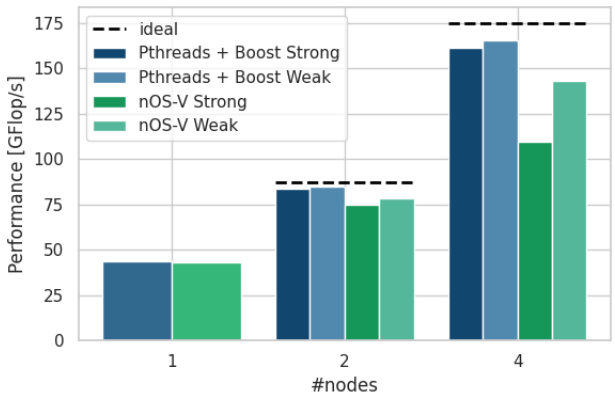

Figure 11. Strong and weak scaling of the Jacobi example using up to 4 nodes of dual-socket Intel Xeon Gold 6238T CPUs. Each node has 22 cores per socket with hyperthreading enabled (but here unused).

The benefits of nOS-V may hence be exploited without noticeable performance impact.

We also evaluate the impact of the computing backend choice on scaling and communication overheads on up to 4 nodes, where each node consists of the same dual-socket system previously used. To run this test, we scale the number of HiCR instances with the number of nodes where each instance handles one rectangular cuboid resulting from splitting the mesh in $p_x \times p_y \times p_z$ parts, where $p_x p_y p_z = p$ the total number of nodes used. After each iteration, halos are exchanged via distributed-memory communication through the LPF Infiniband backend from Section 5.1. We observe the best performance using a $p \times 1 \times 1$ node mesh, which, combined with the thread mesh from the previous experiment, results in a logical $p \times 2 \times 22$ mesh. Fig. 11 shows the resulting weak and strong scaling for both variants. For weak scaling, we increase the number of elements to 880^3 for $p = 2$, and to 1056^3 for $p = 4$.

Pthreads+Boost consistently achieves a better performance than nOS-V in all cases, which, after analysis, we attribute to the use of nOS-V resulting in eager polling of the completion status of distributed-memory communication. This, in turn, causes threads interfering with one another during the communication phase.

6 Conclusions and Future Work

The latest developments in distributed heterogeneous technologies present new challenges for programmers. Developing applications in a way that they can adapt to and stay current with the latest and upcoming hardware can be a daunting endeavor. We have presented a model to minimize this effort, enabling the expression of program semantics that are independent from specific underlying system technologies, bypassing the complexities of writing technology-specific code. Contrary to other solutions, HiCR describes a set of abstract operations without prescribing particular programming models or paradigms, and without making any other implementation decisions. These facets make HiCR a suitable model for a thin Runtime Support Layer that resides between DSLs, libraries, and programming frameworks on the one hand, and raw system technologies on the other.

We have shown empirically that HiCR programs can execute on multiple architectures and employ different supporting libraries, simply by switching out the underlying HiCR backends– and that doing so preserves the overall program semantics. While these experiments also show that applications may be implemented directly on top of the HiCR model, we posit its use as a Runtime Support Layer is of higher value.

Future work includes extending the model for discovery of the interconnect topology, associating latency and bandwidth capabilities to both memory spaces (e.g., in NUMA systems) and interconnect links [57], and the inclusion of (distributed) file management, multi-user job allocation, fault tolerance, and security isolation.

References

[1] 2011. Partitioned Global Address Space (PGAS) Languages. In *Encyclopedia of Parallel Computing*, David A. Padua (Ed.). Springer, 1465.

[2] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguade, and Jesus Labarta Mancho. 2022. OmpSs-2@ Cluster: Distributed memory execution of nested OpenMP-style tasks. In *European Conference on Parallel Processing*. Springer, 319–334.

[3] 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.

[4] AMD Heterogeneous-computing Interface for Portability (HIP) 2025-03-24. . https://rocm.docs.amd.com/projects/HIP/en/latest/

[5] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. 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)*, Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra (Eds.). Springer, 298–299.

[6] 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)*, Henk J. Sips, Dick H. J. Epema, and Hai-Xiang Lin (Eds.). Springer, 863–874.

[7] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. 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)*, Henk J. Sips, Dick H. J. Epema, and Hai-Xiang Lin (Eds.). Springer, 851–862.

[8] David Beckingsale, Thomas R. W. Scogland, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, and Brian S. Ryujin. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC*. IEEE, 71–81.

[9] Roberto Belli and Torsten Hoefler. 2015. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *2015 IEEE International Parallel and Distributed*

Processing Symposium. 871–881. https://doi.org/10.1109/IPDPS.2015.30
[10] Boost Context 2025-03-24. . https://www.boost.org/doc/libs/1_84_0/libs/context/doc/html/index.html
[11] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.
[12] C++ Accelerated Massive Parallelism (C++ AMP) 2025-03-24. . https://learn.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-overview?view=msvc-170
[13] 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.
[14] Paul Cardosi and Bérenger Bramas. 2023. Specx: a C++ task-based runtime system for heterogeneous distributed architectures. *CoRR* abs/2308.15964 (2023).
[15] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 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*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 519–538.
[16] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, Wei Liu, Linfeng Li, Fangming Liu, and Kun Tan. 2024. YuanRong: A Production General-purpose Serverless System for Distributed Applications in the Cloud. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 843–859.
[17] Data Parallel C++ 2025-03-24. . https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html
[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[19] 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.
[20] Jiri Dokulil, Martin Sandrieser, and Siegfried Benkner. 2016. Implementing the open community runtime for shared-memory and distributed-memory systems. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE, 364–368.
[21] 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.
[22] 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*, Minyi Guo and Zhiyi Huang (Eds.). ACM, 1–10.
[23] 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.
[24] 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)*, Hironori Kasahara and Keiji Kimura (Eds.). Springer, 96–111.
[25] 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.
[26] 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.
[27] Roberto Gioiosa, Burcu Ozcelik Mutlu, Seyong Lee, Jeffrey S. Vetter, Giulio Picierro, and Marco Cesati. 2020. The Minos Computing Library: efficient parallel programming for extremely heterogeneous systems. In *Annual Workshop on General Purpose Processing using Graphics Processing Unit*, Adwait Jog, Onur Kayiran, and Ashutosh Pattnaik (Eds.). ACM, 1–10.
[28] Huawei Ascend Computing 2025-03-24. . https://e.huawei.com/en/products/computing/ascend
[29] Huawei Ascend Computing Language 2025-03-24. . https://www.hiascend.com/document/detail/zh/canncommercial/700/overview/index.html
[30] Infiniband Verbs API 2025-03-24. . https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17/vpi+verbs+api
[31] Introduction to Infiniband 2025-03-24. . https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf
[32] Herbert Jordan, Philipp Gschwandtner, Peter Thoman, Peter Zangerl, Alexander Hirsch, Thomas Fahringer, and Dietmar Fey. 2020. The allscale framework architecture. *Parallel Comput.* 99 (2020), 102648.
[33] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 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*, Allen D. Malony and Jeff R. Hammond (Eds.). ACM, 6:1–6:11.
[34] 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.
[35] 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*, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, 115–120.
[36] Light Parallel Foundations - Source Code, Branch: noc_extension [n. d.]. https://github.com/Algebraic-Programming/LPF/tree/noc_extension/. (2025-03-24).
[37] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.
[38] Timothy G. Mattson, Romain Cledat, Vincent Cavé, Vivek Sarkar, Zoran Budimlic, Sanjay Chatterjee, Joshua B. Fryman, Ivan Ganev, Robin Knauerhase, Min Lee, Benoît Meister, Brian Nickerson, Nick Pepperling, Bala Seshasayee, Sagnak Tasirlar, Justin Teller, and Nick Vrvilo. 2016. The Open Community Runtime: A runtime system for extreme scale computing. In *IEEE High Performance Extreme Computing Conference*. IEEE, 1–7.
[39] MNIST Dataset 2025-03-24. . https://www.kaggle.com/datasets/hojjatk/mnist-dataset
[40] MPI Forum 2025-03-24. . https://www.mpi-forum.org/
[41] Nvidia CUDA 2025-03-24. . https://developer.nvidia.com/cuda-toolkit
[42] Nvidia Thrust 2025-03-24. . https://developer.nvidia.com/thrust
[43] Obtuse but Versatile Nanoscale Instrumentation (ovni) 2025-03-19. . https://ovni.readthedocs.io/
[44] OmpCluster 2025-03-24. . https://ompcluster.gitlab.io/
[45] OmpSs-2 2025-03-24. . https://pm.bsc.es/ompss-2
[46] Open Accelerators (OpenACC) 2025-03-24. . https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf
[47] Open Computing Language (OpenCL) 2025-03-24. . https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html
[48] Open Multi-Processing (OpenMP) 2025-03-24. . https://www.openmp.org/specifications/
[49] OpenBLAS: An optimized BLAS library 2025-03-24. . https://www.openmathlib.org/OpenBLAS/

[50] Paraver: a flexible performance analysis tool 2025-03-19. . https://tools.bsc.es/paraver

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

[52] POSIX Threads 2025-03-24. . https://man7.org/linux/man-pages/man7/pthreads.7.html

[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] 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.

[55] Marc Sergent, Célia Tassadit Aitkaci, Pierre Lemarinier, and Guillaume Papauré. 2019. Efficient notifications for MPI one-sided applications. In *Proceedings of the 26th European MPI Users' Group Meeting* (Zürich, Switzerland) *(EuroMPI '19)*. Association for Computing Machinery, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3343211.3343216

[56] 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.

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

[58] SYCL 2025-03-24. . https://www.khronos.org/sycl/

[59] 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.

[60] 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)*, Ramin Yahyapour (Ed.). Springer, 291–303.

[61] 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)*, Ricardo Chaves, Dora B. Heras, Aleksandar Ilic, Didem Unat, Rosa M. Badia, Andrea Bracciali, Patrick Diehl, Anshu Dubey, Oh Sangyoon, Stephen L. Scott, and Laura Ricci (Eds.). Springer, 445–455.

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

[63] XcalableACC 2025-03-24. . https://xcalablemp.org/XACC.html

[64] XcalableMP 2025-03-24. . https://xcalablemp.org/index.html

[65] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*. 273–283. https://doi.org/10.1109/BigData.2016.7840613

[66] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. 2019. DuctTeip: An efficient programming model for distributed task-based parallel computing. *Parallel Comput.* 90 (2019).

[67] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*.