# A Survey of Distributed Asynchronous Many-Task Models and Their Applications

JOSEPH SCHUCHART, Stony Brook University, Stony Brook, United States
PATRICK DIEHL, Los Alamos National Laboratory, Los Alamos, United States
MICHAEL BAUER, NVIDIA Corp, Santa Clara, United States
AURELIEN BOUTEILLER, Advanced Micro Devices, Inc., Austin, United States
GREGOR DAISS, Los Alamos National Laboratory, Los Alamos, USA
ENGIN KAYRAKLIOGLU, Hewlett Packard Enterprise, Palo Alto, United States
SHREYAS KHANDEKAR, Hewlett Packard Enterprise, Palo Alto, United States
THOMAS HERAULT, National Institute for Research in Digital Science and Technology, Paris, France
JOHN HOLMEN, Oak Ridge National Laboratory, Oak Ridge, United States
RITVIK RAO, University of Illinois at Urbana-Champaign, Urbana, United States
ALEXANDER STRACK, University of Stuttgart, Stuttgart, Germany
ELLIOTT SLAUGHTER, SLAC National Accelerator Laboratory, Menlo Park, United States
JENNIFER SPINTI, The University of Utah, Salt Lake City, United States
JEREMY THORNOCK, Lawrence Livermore National Laboratory, Livermore, United States
ALEX AIKEN, Stanford, Stanford, United States
OLIVIER AUMAGE, National Institute for Research in Digital Science and Technology, Le Chesnay, France
MARTIN BERZINS, The University of Utah, Salt Lake City, United States
GEORGE BOSILCA, NVIDIA Corp, Santa Clara, United States
BRADFORD CHAMBERLAIN, Hewlett Packard Enterprise, Palo Alto, United States
HARTMUT KAISER, Louisiana State University, Baton Rouge, United States
LAXMIKANT KALE, University of Illinois at Urbana-Champaign, Urbana, United States

Asynchronous many-task (AMT) runtime systems have become an important paradigm for expressing fine-grained parallelism and managing asynchrony in high-performance computing (HPC). Originating from early dataflow concepts, AMTs have evolved to enable dynamic task generation, explicit dependency management, and asynchronous execution, facilitating the overlap of computation and communication. These capabilities address the limitations of traditional bulk-synchronous models, such as those employed in MPI+X, which can struggle with irregular, adaptive, or data-driven workloads. This survey provides a comprehensive overview of representative distributed AMT systems—including Charm++, HPX, Legion, PaRSEC, Uintah, Chapel, and StarPU—focusing on their design principles, execution models, and runtime mechanisms for scheduling, communication, and synchronization. We examine how these systems tackle key challenges such as load imbalance, runtime overheads, programmability, and performance portability. In addition, the paper discusses application domains where AMTs




ACM 1557-7341/2026/8-ART
https://doi.org/10.1145/3840389

have demonstrated tangible benefits and highlights the conditions under which their use is most advantageous. The goal of this survey is to equip researchers and practitioners with a clear understanding of distributed AMT models and to provide guidance for selecting and applying the most suitable runtime system for specific computational objectives. [1]

CCS Concepts: • Computing methodologies → Distributed computing methodologies; Parallel algorithms.

Additional Key Words and Phrases: Asynchronous many-task models, distributed, applications

## 1 Introduction

The idea of asynchronous many-task (AMT) models started in the 1980s around the concept of dataflow. Here, the computation was driven by the availability of data and not sequential code execution. MIT's Id language [1] and T-dataflow architecture [2] are two examples. In the 1980s and 1990s, a transition to software-managed concurrency occurred. Implementations of parallel functional and logic languages gave rise to underlying AMT systems, such as RediFlow [3] and Chare Kernel [4]. Cilk [5] provided a thread-based, structured, task-based parallelism addressing dynamic load balancing via work-stealing. Cilk supported efficient runtime scheduling and fork-join semantics to express parallelism. Another approach proposed by Charm++ [6–8] is message-driven execution and migratable objects, namely chares, supporting adaptivity and asynchronous execution. Charm++ was one early adopter of over-decomposition to address load balancing and latency hiding. In the 2000s, the focus was on dynamic task graphs, fine-grained tasks, and event-driven execution. These features were designed to overlap communication and computation and maximize resource utilization. Here, HPX [9], Chapel [10], StarPU [11], Go [12], OpenMP, Uintah [13], Qthreads [14], and Cilk Plus [15] were released. In the 2010s, PaRSEC [16], Legion [17], Rust, UPC++ [18], and Julia [19] were released.

While the historical evolution of AMTs offers valuable insights, it is equally important to consider the underlying motivation: what challenges led to the emergence of AMTs? The Message Passing Interface (MPI) [20] has long served as the lingua franca for inter-process communication in high-performance computing (HPC). The term MPI+X refers to a hybrid programming model in which MPI is responsible for inter-node communication, while some other X—such as OpenMP, CUDA, or Kokkos—handles intra-node parallelism. This approach leverages the strengths of MPI for distributed memory systems and complements it with node-level parallelism frameworks tailored to shared memory or accelerator-based architectures. The MPI model follows a hierarchical structure and typically assumes: a) a bulk-synchronous execution model, b) explicit specification of both parallelism and data movement, and c) application-level management of dependencies and synchronization. For certain applications—such as adaptive mesh refinement (AMR), particle-based methods, and graph-based computations—the assumptions outlined in a)–c) can pose significant challenges to performance and scalability.

This survey is limited to distributed AMT. Let us take a closer look at how distributed AMT models address these challenges: a) Rather than relying on bulk synchronization, AMT systems enable fine-grained task scheduling and dependency management, allowing tasks to execute independently as soon as their inputs are ready. This approach minimizes idle time, mitigates load imbalance, and improves resource utilization, which is particularly beneficial for irregular or dynamic workloads. b) To avoid the explicit specification of parallelism and communication, AMT models leverage implicit concurrency and task-

or data-driven execution. Computation is expressed in terms of tasks, and the application developer specifies only the interactions among tasks, not how or when they execute. c) Typically inter-node communication is not explicitly programmed; instead, the AMT runtime analyzes task dependencies and data access patterns to automatically orchestrate data movement across nodes or memory spaces. Many AMT systems further improve efficiency by overlapping communication with computation through asynchronous execution. We note that some AMT systems still use MPI for communication but hide its usage from the application developer. d) some AMT systems allow the programmer to omit specifying which nodes/processors will execute particular tasks.

This paper first presents a survey of seven representative AMT models, examining how each implements the key features outlined above. Subsequently, we analyze the challenges inherent in the AMT paradigm, including those related to programmability, performance portability, and runtime complexity. Finally, we highlight representative scientific applications for each of these AMTs, illustrating where their use is advantageous and providing guidance on the types of workloads and characteristics that benefit from AMT models, as well as those for which they may be less effective.

Goal of the survey paper. This survey aims to guide readers in understanding and selecting suitable distributed AMT runtime systems for their specific use cases. First, we present a comparative analysis of the key features and architectural distinctions of existing distributed AMTs, enabling readers to assess which solutions align best with their requirements. Second, we explore a range of application domains where distributed AMTs have demonstrated tangible benefits, highlighting patterns of adoption and performance gains to support informed decision-making.

Structure. The paper is structured as follows: Section 2 covers related work. Section 3 defines the terminology used in this paper. Section 4 provides an overview of all participating distributed AMTs. Section 5 discusses the challenges and opportunities of AMTs. Section 6 showcases applications that used the surveyed AMTs. Section 7 discusses application characteristics for which AMTs are beneficial. Finally, Section 8 concludes the work.

## 2 Related work

This work focuses on distributed AMTs, many of which are surveyed in this work. However, two notable exceptions are worth mentioning: (1) UPC++ [18] provides an asynchronous communication framework using Remote Procedure Call (RPC) and Partitioned Global Address Space (PGAS) and (2) DARMA/VT [21] supports a distributed AMT programming model while seamlessly integrating with MPI.

Notable AMTs with shared memory support are Cilk Plus [15], which extends the C++ programming language and adds parallel for loops and fork-join support; Intel© Thread Building Blocks (TBB) [22], now Intel© oneAPI TBB, which is a C++ template library which supports tasks that can be run in parallel; Qthreads [14], which is a lightweight, locality-aware, user-level threading runtime; Rust, which provides asynchronous programming and futures; Go [12], which provides concurrency via goroutines similar to coroutines; OpenMP [23], which is a widely used language extension for C/C++ and Fortran that provides—among other things—task dependencies for shared memory and GPU programming. Note there are efforts to extend OpenMP tasking for distributed computing [24, 25]; OmpSs [26] and OmpSs2 [27], which extend OpenMP with new directives to support asynchronous parallelism and heterogeneity; TaskTorrent a parallel runtime library for executing concurrent directed acyclic graphs of computational tasks based on concepts presented in 1995 [28]; SPECX shares several similarities with StarPU but is written in modern C++ [29]; IRIS, which is a portable runtime system exploiting multiple heterogeneous

programming systems [30]; Julia [19], which provides tasking via coroutines and channels; Taskflow [31] is a C++ library that provides control-flow synchronization between tasks; and nOS-V [32], a low-level threading and tasking library that enables co-execution of different runtime systems, was released.

Note that most of these can be used with the MPI+X paradigm; however, communication is done explicitly by the user via the MPI API and is not automated by the AMT as with the other AMTs reviewed in this paper. For a detailed comparison, we refer to [33] and for a comparison on software metrics to [34].

There are notable distributed memory AMTs past and present. IBM©'s X10 [35] appears to be discontinued. XcalableMP (XMP) [36] is a directive-based language extension and provides an interface to MPI for distributed programming. However, the latest release of the specification, 1.4, was done in 2018. Control-flow dependencies have been proposed as an extension to the Partitioned Global Address Space (PGAS) model [37]. Itoyori ( ) is a distributed multithreading runtime system for global-view, fork-join task parallelism [38]. Julia's Dagger offers a distributed with automatic data dependency analysis [39]. Other research efforts such as HTGS/Hedgehog from NIST have shown great promise in delivering high performance on GPUs [40]. CUDA-STF provides tasking for NVIDIA GPUs [41].

While the AMTs listed above are primarily focused on HPC, it is worth noting that many other AMTs exist in the domains of data science, data analysis, and machine learning. These alternatives, however, differ significantly in their characteristics. They typically incur substantially higher overhead compared to HPC-focused systems and are designed to support different capabilities—such as elasticity and resilience—which are often essential in cloud-native environments. Despite these differences, their relevance merits mention. These include MapReduce [42], Ray [43], Spark [44] (data analytics), PyTorch [45], and Jax [46] (machine learning).

Task Bench[2] is an effort to evaluate the efficiency and performance of parallel and distributed programming models, runtimes, and languages. Many of the models discussed in the related work and surveyed in this work are compared here [47, 48]. The current state and micro benchmarking is presented in [49].

Figure 1 shows the evolution of AMTs with Charm++ as one of the earliest ones and UPC++ being one of the newer ones. The large cluster from 2006-2012 indicates a golden era of AMTs whereas fewer new AMTs have appeared recently.

## 3 Terminology

In this section, we define common terms used to describe the semantics of these models.

Tasks. Tasks are work packages that are typically executed as asynchronous activities with defined data input and output. In general, tasks are either required or assumed to be free of any side-effects, except on their declared arguments, operating on those arguments to produce one or more results.[3] Tasks can be created explicitly by the application through a spawn function or implicitly as part of a broader algorithm. Tasks are managed by a runtime system, which assigns the execution of individual tasks to execution resources. The execution of a task may lead to the discovery of new tasks; i.e., tasks may, in general, dynamically launch subtasks. In some models, the output of a task will become the input of successive tasks, forming a directed acyclic graph (DAG) of tasks. Tasks are typically used to oversubscribe available resources to hide imbalances and communication latencies.

[2] https://github.com/StanfordLegion/task-bench

[3] In practice, however, some models allow tasks to access global and thread-private variables but do not provide any ordering based on these accesses unless they are visible to the runtime system.

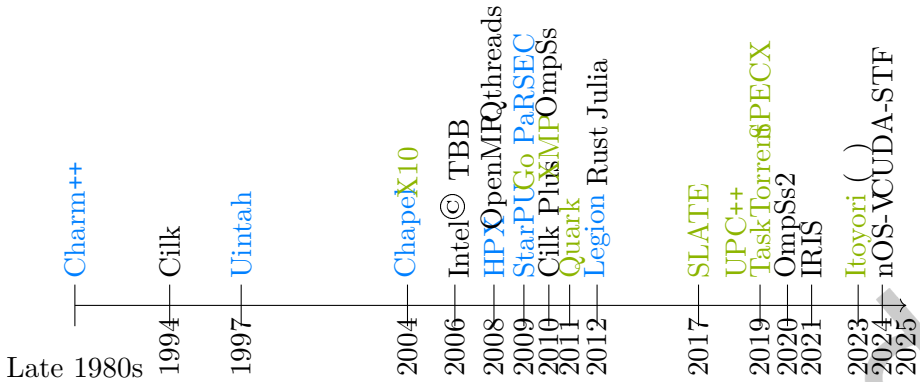


Fig. 1. Timeline of the AMTs. The blue-colored AMTs are covered in this study and provide distributed capabilities. The green-colored AMTs also has distributed capabilities but is not part of the survey. All other AMTs only provide shared memory parallelism. Most of them can be extended to distributed runs using the *MPI+X* paradigm. For the date, we chose either the first publication mentioning the AMT or the first open source release.

Execution Domains. Distributed AMTs generally utilize multiple distinct hardware nodes with one or more CPUs and GPUs.[4] The runtime system itself typically runs on one or more CPUs on each node and controls the resources of that node. Tasks are commonly assigned a discrete set of resources by the runtime system, such as one or more CPU cores and GPUs, to perform the work of the task. As shown in Figure 2, the host and GPU form distinct execution spaces.

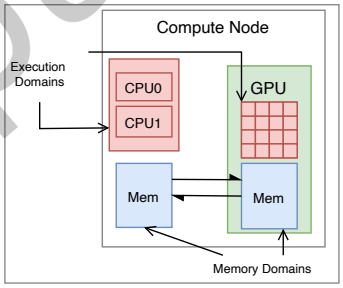


Fig. 2. Memory and execution domains.

Memory Domains. Separate execution domains typically utilize separate data domains, e.g., host memory for CPU cores and GPU memory for computations executed on kernels. When executing across multiple physical nodes, the memory of these nodes can be viewed as separate domains or combined into a single domain accessible by all processes, e.g., using the PGAS [50] paradigm. Figure 2 shows separate memory domains with explicit transfers between host and device memory.

Data Dependencies. Dependencies are a mechanism for describing the synchronization and potential dataflow between tasks. In their basic form, they describe the input and output of tasks based on existing (distributed) data structures. Dependencies are a backward-looking mechanism where input/output dependencies declared on a task synchronize that task's execution with previously discovered tasks that have declared conflicting dependencies on the same data, expressing required happens-before relations between tasks. In the example of Listing 1, a number of $N$ tasks are submitted to the runtime, with an `OUT` dependency specifying that the task will write a value to the

Listing 1. Data dependency example computing N values and reducing them to a single value.

```
vals[N], val
for (i in 0..N-1)
  add_task(fn:{vals[i]=eval(i)}},
           OUT:vals[i])
add_task(fn:{val=reduce(vals,+)},
         IN:vals[0..N], OUT:val)
fence()
print(val)
```

[4]Though accelerators other than GPUs are also of potential interest, we limit the scope of our discussion to GPUs. At the time of this writing, none of the AMTs discussed in this paper support upcoming accelerators with alternative architectures.

position in array `vals`. A second task declares `IN` dependencies on each element in the array and reduces them into a single value `val`. The resulting task-graph is depicted in Figure 3.

Dependencies may provide data-movement or synchronization-only capabilities. For example, while OpenMP task dependencies provide only synchronization in shared memory, some models described in this article are dataflow models that utilize dependencies to express the required data movement in distributed memory.

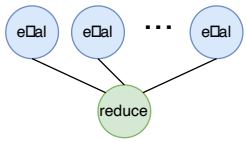


Fig. 3. Task graph resulting from the examples.

Futures. Futures (such as `std::future` introduced in C++11) are a particular approach for moving data from a producing to a consuming task. Futures are handles to a value (or set of values) that will be generated by a task. They can be used to wait for and access the resultant value. Such a value is set by the producer via a promise. The connection between a promise and a future forms a unidirectional channel that connects the producer to the consumer. Futures can be used by the runtime system to guide scheduling decisions, either by suspending a task waiting for a future or by deferring the execution of a task until all input futures have been fulfilled. Listing 2 shows the same task-graph depicted in Figure 3 expressed using futures.

Listing 2. Listing 1 expressed using futures.

```
fut fs[],f2
for (i in 0..N-1)
  fs[i] = async(fn:{ret eval(i)})
f2 = whenall(fs).then(
       fn:(vals){
         ret reduce(vals,+)
       })
print(f2.get())
```

Parameterized Task Graphs. Parameterized task graphs express an algorithm as a set of task classes with inputs and outputs that are connected explicitly. During the execution of the algorithm, task classes are instantiated into tasks, with data flowing between them. The completion of one task will lead to the discovery of successor tasks, potentially dependent on the result itself. In such a system, the runtime has a priori information on all possible tasks and the flow of data between them. At runtime, only a subset of these tasks may be instantiated, based on the parameters of the algorithm and the results of each task. Listing 3 demonstrates the use of a parameterized task graph where the flow of data is expressed through arrows connecting inputs and outputs of task classes (`EVAL`, `REDUCE`, and `PRINT`).

Listing 3. Listing 1 expressed using parameterized tasks.

```
EVAL(k=0..N-1) : val->REDUCE[k]
{ val = eval(k) }
REDUCE()
: vals[k]<-EVAL(k),val->PRINT()
{ val = reduce(vals, +) }
PRINT() : val<-REDUCE()
{ print(val) }
```

Parallel Algorithms. Typical algorithms used in computing include sorting, searching, accumulating, transforming, and iterating. Parallel algorithms (as introduced in C++17) are higher-level abstractions for such computing patterns that execute the algorithm in parallel from a single entry point. In the case of AMTs, an implementation may spawn a number of tasks that are orchestrated to implement the semantics of the specific algorithm. For example, parallel accumulation over a set of values may spawn a DAG of tasks that resemble a binary tree through

Listing 4. Listing 1 expressed using parallel algorithms.

```
fut f1,f2
f1 = par_transform(0..N-1,
       fn: (i){ret eval(i)})
f2 = f1.then(
       fn:(vals){
         ret reduce(vals,+)
       })
print(f2.get())
```

which the initial and intermediate values are accumulated to
arrive at the result at the root of the tree.

Parallel algorithms simplify the use of AMTs by shifting the responsibility of expressing concurrent operations from the application to the implementation of the algorithm. They can form the building blocks that can be composed into more complex algorithms. However, not all algorithms can be efficiently expressed solely as a composition of parallel algorithms.

Parallel Composition. A programming system is said to support parallel composition when multiple independently developed algorithms, each running on most/all/many execution domains (nodes and processors of the system), are able to interleave their execution on individual execution domains. In the absence of this property, the system or the programmer is reduced to either running the algorithms one after the other or running them on disjoint execution domains. Notably, MPI does not support such parallel composition. AMT systems have a potential to support it because tasks are executed based on availability of data, thus permitting the interleaving of execution.

Performance Portability. The proliferation of various GPU programming models (e.g., CUDA, HIP, SYCL, OpenCL, and OpenMP) requires constant adaptation of applications to be able to fully utilize the computational resources provided by new architectures. Approaches for performance portability such as Kokkos [51] and RAJA [52] provide a restricted set of parallelization primitives that can be mapped to all the supported architectures without modifications to the application code. Not all AMTs covered in this article provide full-stack performance portability. Some focus only on data movement and the orchestration of tasks while relying on applications to cover intra-task parallelism (e.g., within a GPU kernel).

Load Balancing. Load imbalances are a common occurrence in parallel applications, stemming from hardware inconsistencies [53], operating system (OS) noise [54], non-uniform data distribution across processes, and shifts in workload between processing elements inherent to the algorithm. In order to maximize the utilization of hardware resources, runtime systems may track and balance the load within a process and across processes. The oversubscription of resources within a task-based application enables load balancing across CPU cores within a single process, i.e., intra-process. Runtime systems may also balance the load between multiple accelerators controlled by a single process by moving data between devices. Some AMTs provide distributed (i.e., inter-process) load balancing; however, a trade-off exists between the benefits of load-balancing and the potential costs incurred by tracking load at a global scale as well as the required data movement and constraints on tasks (i.e., absence of side-effects).

## 4 Overview of AMTs

This section provides an overview of all distributed AMTs surveyed. The features of each AMT are described concisely on one page and summarized in Tables 1 and 2. The timeline in Figure 1 shows the first occurrence of the AMT; the length of existence may affect feature completeness. The following AMTs are surveyed: 4.1 Chapel, 4.2 Charm++, 4.3 HPX, 4.4 Legion, 4.5 PaRSEC, 4.6 StarPU, and 4.7 Uintah.

### 4.1 Chapel

Chapel [10, 55, 56] is a scalable parallel programming language made available under the Apache 2.0 license and is part of the High Performance Software Foundation. The name Chapel derives from the moniker 'Cascade High Productivity Language.' Chapel is a compiled, statically typed language designed for productive parallel computing at scale. It aims to improve the programmability of parallel and distributed computers while matching the performance and portability of existing models. Chapel features

a modern syntax, similar to that of Python, yet its performance is as good as Fortran or C++. In addition, Chapel applications have scaled to thousands of nodes and over a million cores, competing with MPI and SHMEM [57, 58].

Unlike other programming models built on top of existing languages or frameworks, Chapel was developed from the ground up to help programmers reason about their algorithms with features designed to support scalable parallelism in a natural and optimizable manner.

Chapel supports a high-level fork-join model for general task parallelism [59]. All Chapel programs begin as a single task, but the language provides three primary constructs for introducing additional tasks: `begin`, `cobegin`, and `coforall`. The `begin` statement spawns a single, fully asynchronous task, while `cobegin` allows a fixed number of tasks that must all complete before the original task continues. The `coforall` loop enables a loop-based approach to task creation, where each iteration runs the loop body as its own independent task. These constructs can be nested to introduce arbitrary and asynchronous task-based parallelism. Tasks coordinate and synchronize through variables in their shared lexical scopes, including `sync` variables that support full-empty semantics and `atomic` variables supporting the standard set of atomic operations.

Chapel's design incorporates a global view, enabling users to reason about their code's execution context through the concept of locales. A locale typically refers to a node or socket in a distributed system. Execution starts on Locale 0, and users can move tasks to different locales using `on` statements. This is enabled by Chapel's compiler and runtime, which support migrating tasks to remote locales. Chapel supports a lexically based partitioned global namespace in which tasks can read and write data on remote locales by referring to variables and arrays in shared scopes. These operations are implemented in Chapel's runtime using active messages and one-sided PUTs and GETs. Unlike traditional SPMD programming models, in which the programmer codes from the perspective of a single process, Chapel's global view allows for more intuitive expression of distributed parallel applications. That said, Chapel users can also create an SPMD-style region by combining a `coforall` loop with an `on` statement.

Chapel also features `forall` loops that support high-level data parallelism. When used in conjunction with Chapel's distributed arrays, `forall` can spawn tasks globally across the system wherever the data resides, enabling easier expression of distributed computation and resource utilization. The `forall` loop relies on the distribution that is being iterated over to specify how and where to spawn tasks. Notably, all `forall` loops and distributed arrays are implemented within the language itself by writing distributed data structures and parallel iterators that are defined using lower-level tasking constructs like `coforall` loops and `on` statements. Users can define their own custom parallel iterators and/or arrays, allowing additional flexibility in how parallelism is implemented. Chapel also supports programming GPUs using the same `forall` loops to create GPU kernels in a vendor-neutral manner [60, 61]. This capability enhances the power and expressiveness of Chapel, allowing developers to tailor parallelism to their specific needs.

### 4.2 Charm++

Charm++ [6–8] is a C++-based parallel programming system based on migratable objects. A Charm++ computation comprises one or more individually indexed collections of a special type of C++ objects called Chares that can be migrated between nodes/PEs. A chare has entry methods, which may be asynchronously invoked from any processor. A chare is uniquely identified by its collection ID and its index within that collection, without any need for the callers to know its location. An Entry method of a chare can usually only access its own data directly (except global read-only data for example) and is allowed to run to completion, without preemption, unless it decides to suspend. No two methods of a

chare can execute concurrently. A notation, structured dagger [62], (SDAG) allows expression of task dependencies.

Consider LeanMD [5], a Lennard-Jones molecular dynamics mini-application (Figure 4). Atoms are partitioned into a 3-dimensional dense collection (aka array) of chares called cells and a 6-dimensional sparse array of chares calculates force interactions between nearby pairs of cell chares. The programmer specifies interactions between members of these chares, without concerning themselves with the processor they are housed on.

A user-space scheduler selects method invocations (tasks) from queues, with optional priorities. A distributed location management module tracks current locations of chares and forwards messages when needed. Since the runtime system is mediating communication and scheduling chares, it is able to track computation loads and communication patterns, which is used by its dynamic load balancers to migrate chares across nodes/PEs to balance loads and minimize communication.

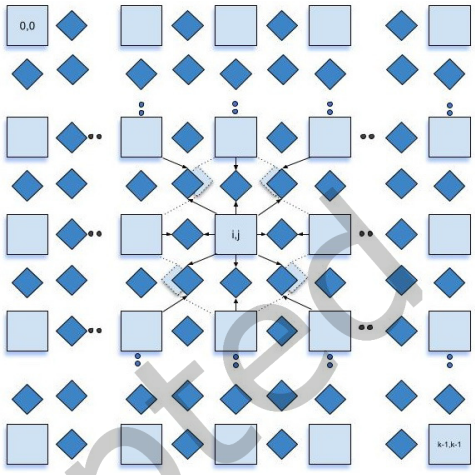


Fig. 4. LeanMD in 2 dimensions, with a kxk array of cell chares and a kxkx3x3 sparse array of pair chares

Overdecomposition (using many more chares than the number of PEs), asynchrony (due to message-driven scheduling) and migratability are the key features of Charm++. They empower its intelligent runtime system to support unique capabilities such as (a) distributed dynamic load balancing, (b) automatic and adaptive overlap of communication and computation, (c) energy optimizations for sustainability, where chip temperature and power are constraints while energy and execution time are metrics to optimize [63], and (d) multiple fault tolerance schemes [64], including a practical in-memory checkpoint with an automatic failure detection/restart scheme. Migratability can be leveraged to support elasticity [65], which is the dynamic shrinking and expansion of Charm++ programs to fit available resources, which is especially useful in cloud environments.

Parallel composition of algorithms: Independently written parallel algorithms, each using all processors, can run concurrently, interleaving their execution on individual processors because of its message-driven scheduler.

Evolution: There have been many architectural changes over 24 years of modern Charm++ [6]. For example, Charm++ supports asynchronous orchestration of accelerator tasks, via device-specific kernels or Kokkos, while Torus topologies led to a new class of load balancers. Charm++'s basic parallel model has remained largely invariant, which attests to the resilience of its foundational concepts.

Ecosystem: Charm++ comes with a pioneering visual performance analysis system (Projections [66]), and a record-replay-based debugging system. Adaptive MPI [67] is an MPI implementation atop Charm++ that virtualizes the notion of MPI ranks. Several higher-level languages have been implemented [68] on top of Charm++ including Charisma, Multiphase shared arrays, and divCon. Charm4Py [69] is an implementation of the Charm++ model in Python. CharmNumerics, a Numpy-style library, is being developed as part of a larger project called charmTyles, which focuses on interactive computing in

[5] https://charmplusplus.org/miniApps

Charm++. We believe that charmTyles [70] can act as an ideal back-end for HLLs, automatically providing features such as load balancing.

### 4.3 HPX

HPX [9, 71] is a high-performance asynchronous many-task (AMT) runtime system, released under the Boost Software License, and is part of the High Performance Software Foundation (HPSF). It delivers flexibility, efficiency, portability, and scalability up to ExaScale [72]. HPX offers a C++-standard conforming API, extending Futures to support distributed and heterogeneous computing, enabling seamless local and remote parallelism.

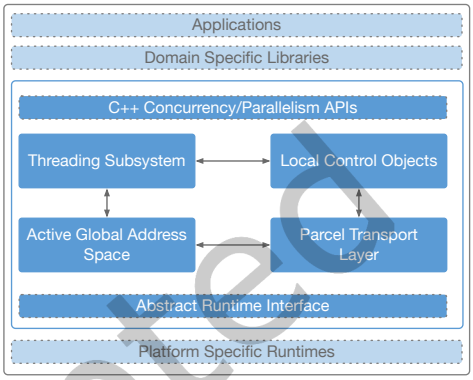


Fig. 5. HPX's architecture from the application level down to the hardware level.

HPX is the first implementation of the ParalleX execution model [73], built from the ground up for scalability. It efficiently exploits computational resources by minimizing overhead, latency, and contention. HPX supports fine-grained asynchronous parallelism, scaling to millions of lightweight tasks in shared- and distributed-memory systems, with support for Nvidia, Intel, and AMD GPUs via Kokkos [51]. The integration is two-fold: 1) HPX-Kokkos allows for asynchronously launching Kokkos functions [74], and 2) HPX as a Kokkos CPU backend. Its abstraction of low-level hardware ensures performance portability across diverse heterogeneous platforms, including A64FX [75] and RISC-V [76, 77].

Architecture. Figure 5 illustrates the core components of HPX that underpin its capabilities. The Threading Subsystem manages lightweight threads for fine-grained parallelism; AGAS (Active Global Address Space) offers a unified view of distributed memory; the Parcel Transport Layer enables efficient inter-node communication; and Local Control Objects support constraint-based thread synchronization.

HPX addresses the demands of large-scale high-performance applications through key features: asynchronous execution and communication with message batching, dynamic load balancing and task partitioning, and portability of code and performance. Its architecture emphasizes efficiency, modularity, and scalability, aiming to deliver a cutting-edge AMT runtime system as a robust foundation for modern parallel applications.

- Active global address space (AGAS): Dynamic load balancing of distributed data is simplified by a unified global address space. AGAS enables transparent migration of global objects across localities without changing their addresses and supports distributed garbage collection for these objects [78].
- Threads and thread management: The HPX thread manager uses a work-queue execution model with work-stealing. Ready tasks are wrapped in lightweight, cooperatively scheduled HPX-threads, reducing context switching and scheduling overhead.
- Message transport and message management: HPX's inter-locality communication relies on Parcels, an extended form of active messages, enabling asynchronous, message-driven control flow and dynamic resource management. Parcels move work to data or retrieve small data fragments to the caller. HPX supports multiple networking backends, including TCP/IP, MPI, LCI [79, 80], OpenSHMEN, and GASNet.
- Local and global synchronization objects: These abstractions support event-driven task creation, asynchronous flow control, race condition protection, and Futures-based synchronization. HPX also

offers standard C++ synchronization primitives (e.g., mutexes, condition variables, semaphores) adapted for cooperative use with HPX-threads, avoiding blockage of underlying kernel threads.
- Performance Counter Infrastructure: HPX provides a unified, extensible API for accessing runtime and application performance data. It supports both built-in and user-defined metrics, enabling adaptive, on-the-fly tuning of algorithms and system parameters [81].

### 4.4 Legion

Legion is a tasking programming model that guarantees sequential semantics while the implementation extracts as much parallelism for performance as possible [17]. The sequential semantics significantly eases development and debugging, as the pitfalls of explicitly parallel programming (such as race conditions and coherence bugs) are not possible. A program begins with its top-level task, which may recursively launch subtasks. Data is organized into multi-dimensional tensors/arrays called regions, following a long history in programming languages of region-based memory management [82, 83] (the name Legion is a contraction of the term logical region).

Tasks are dynamically submitted to the Legion runtime system, which analyzes the task stream on-line for dependencies among the tasks' region arguments and maps the tasks and their regions to the available processor and memory resources. Legion is designed for distributed execution and supports multiple kinds of CPUs and GPUs.

The most radical aspect of the Legion design with respect to other tasking systems is the model for partitioning data. Legion allows multiple overlapping partitions of regions to exist simultaneously. For example, a mesh might be partitioned into contiguous pieces, where each piece $p$ is further partitioned into $p$'s interior points that do not sit on the boundary with another piece, $p$'s boundary points, and $p$'s ghost points, which are boundary points of a different piece that are adjacent to the boundary of $p$. Since one piece's boundary point can be another piece's ghost point, these sets represent different views onto the same data. During execution, the runtime manages access to instances of regions (which may include multiple copies of the same region as well as regions that overlap) to guarantee coherence of the data. The great flexibility of being able to define overlapping, or aliasing, views of data has been crucial to the design and implementation of many Legion applications as well as a source of research challenges [84]. Legion provides an expressive sublanguage of highly performant partitioning operators; by composing these primitives, sophisticated data partitioning strategies can be written in a few lines of code [85].

Legion also takes a novel approach to mapping, exposing almost all performance-relevant decisions in the runtime system to applications through a mapping interface that allows applications to set their own policies rather than relying on heuristics built into the runtime system. Among other options, mappers can exercise full control over the processor on which a particular task is placed, the memories in which the task's region arguments are placed, and how the data for each region is laid out. Importantly, performance decisions made by the mapper never influence the correctness of the application. Multiple mappers can coexist simultaneously in the same application to coordinate execution of tasks from different libraries. Legion comes with a default mapper, so writing and tuning mappers for better performance is optional, and quite recently there has been significant progress on generating high-performance mappers automatically through autotuning [86] and using generative AI [87].

A third important feature is Legion's approach to scale-out parallelism, which, perhaps surprisingly, can be a challenge in task-based systems. Since each task is a single thread of control, launching thousands of tasks from one parent task can be a performance bottleneck. To address this challenge, Legion supports control replication, a novel technique that transparently replicates the execution of a task's control logic on multiple processors (perhaps distributed) while preserving the abstraction that they execute as a single

logical task [88, 89]. The effect is to provide SPMD-style scalability while still guaranteeing sequential semantics.

Legion has a mature and growing set of tools. Regent is a programming language for Legion, providing syntactic and typechecking support for the Legion programming model but also supporting optimized kernel code generation for CPUs and GPUs. The Legion profiler, legion-prof, provides extensive, detailed visualization of the timeline of execution, showing when every operation starts and ends and each task $t$'s critical dependency (of all the operations $t$ depends on, $t$'s critical dependency is the last one to finish and so the primary reason $t$ could not run earlier).

Legion also has a growing set of applications that are described in Section 6.4. All three of these systems heavily leverage the novel data partitioning, mapping, and control replication features discussed above.

### 4.5 PaRSEC

The Parallel Runtime Scheduling and Execution Controller (PaRSEC) is a distributed runtime system that manages task execution and data movement [16]. PaRSEC itself provides a core runtime system and two programming models, the original Parameterized Task Graph (PTG) and the Dynamic Task Discovery (DTD) [90]. PaRSEC is also the main backend the Template Task Graph (TTG) model [91, 92]. These models will be described below.

Initially developed as a runtime for distributed linear algebra (i.e., DPLASMA [93]), PaRSEC is a fully distributed dataflow engine. It provides distributed data structures that can be used to organize an initial data distribution, such as 2D block-cyclic and irregular sparse matrices. Data flows through the task graph and the life time of the objects (and their representations in different data domains) is transparently managed by PaRSEC. Tasks ready for execution will be presented with a valid copy of their input data (that is, the runtime stages data to the target node and device memory implicitly, before enabling the task execution) and the scheduler ensures that there are no conflicting accesses to data, provided the task adheres to its declared access modes.

The PaRSEC runtime system consists of various engines controlling different aspects of the distributed parallel execution (Figure 6). The distributed dataflow engine takes care of data flows between tasks, signaling task completion between nodes, moving the output of one task to the input of another task, managing object life-cycles, communication, and task life-cycle. A separate device execution engine manages the access to accelerators present in the system. Tasks that have a suitable incarnation for a given device type are sent to the device manager, which then inspects the tasks inputs, selects a suitable device, and ensures that all input data is available on that device. The task may then insert the corresponding kernel into a provided stream and the device manager will manage the data movement out of the device (if necessary) once the kernel(s) have completed their execution.

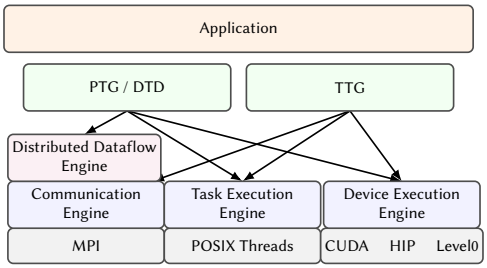


Fig. 6. PaRSEC architecture from the application layer to the hardware level.

The communication engine provides active message and one-sided communication semantics and is currently built on top of MPI. An experimental version using LCI has shown that alternative communication backends are feasible and beneficial [94].

Parameterized Task Graph (PTG): The task graph of an algorithm is expressed through a compact, symbolic language (JDF) where task identifier ranges are defined based on parameters and named output

terminal are connected to named input terminals. This provides the runtime system with full knowledge of the structure of the task graph once the parameters are known at runtime. Despite its scalability, the use of a custom language has posed an adoption challenge for applications, which has driven to the creation of additional input languages for PaRSEC.

Dynamic Task Discovery (DTD): While PTG provides a symbolic structure for task graphs, the DTD model is closer to OpenMP task dependencies and the StarPU STF model (described in Section 4.6). It provides a C API for inserting tasks with dependencies on global data structures. This entails that all processes sequentially discover the task graph, that is then pruned by the runtime system. Communication between tasks on different domains (nodes, accelerators) remain implicit, inferred from the dataflow by the runtime.

Template Task Graph (TTG): While not directly part of PaRSEC, TTG can be considered a direct descendant of PTG. Using C++, it provides a more flexible API that enables data-dependent task graph discovery by allowing tasks to select successors after the computation in a forward-looking model. In contrast to PTG and DTD, TTG does not rely on distributed data structures and instead allows for arbitrary C++ objects to flow through the task graph. Template tasks are objects that define the inputs of tasks, with unnamed input and output terminals connected through named edges. These edges facilitate the composition of task graphs by connecting inputs and outputs of template task graphs [95].

### 4.6 StarPU

StarPU [11, 96–101] is a high-performance AMT runtime system, released under the GNU LGPL v2.1 Software License. It implements the Sequential Task Flow (STF) programming model, in which tasks are submitted by the applicative code in the natural, sequential algorithmic order, and are scheduled and executed in parallel by StarPU in the background, while enforcing data dependence constraints.

On homogeneous nodes, StarPU uses a locality-aware work-stealing scheduling policy to map tasks on computing resources. It takes into account the topology of the node such as NUMA domains and the cache hierarchy to let worker threads attempt to steal works from the most tightly related computing units first.

On heterogeneous, accelerated platforms, StarPU can schedule a kernel task on any computing unit (CPU, GPU, FPGA) for which a kernel implementation is available. A variety of heterogeneous scheduling policies are supplied to decide how to map tasks on the most relevant computing units. Such policies rely on StarPU's performance modeling support to estimate the execution time of a task kernel on the various available computing units given its input data size. Additional components can be taken into account such as data transfer costs to move data from one memory space (for instance, the main memory) to another memory space (for instance, an accelerator embedded memory), or such as user-hinted task priorities.

Data transfers, replication, and consistency management between discrete memory spaces are handled by StarPU's internal distributed shared memory (DSM) in a transparent manner for the application developer. Replication and consistency management services enable StarPU to elide redundant data transfers between memory spaces, by keeping read-only data replicates in embedded accelerators' memories as long as they are not invalidated by a write access, or evicted to make room for other contents.

Granularity management is handled dynamically through StarPU's recursive tasks, together with its splitter component which is in charge of deciding when to transform tasks into sub-graphs at runtime. A linear programming policy [101] aims at minimizing execution time and maximizing resource usage, resulting in a well balanced, heterogeneity-aware workload comprising both small tasks to fill multiple CPU cores, and large tasks for efficient execution on accelerators like GPU devices.

StarPU exposes two models for distributed computing. The first follows a scheme master-worker, where the master process schedules tasks and ensure load balancing on the worker processes. The second one follows a fully distributed scheme, where all participating processes execute a subset of the logical global task graph using a data owner computes rule to allow each process's StarPU instance to decide on task execution entirely locally. In both schemes, distributed data transfers are handled transparently on top of MPI. However, the fully distributed scheme is much more scalable by design, since it avoids the bottleneck of the master process, and since the distributed STF programming model allows for extensively pruning the logical global task graph into thin local task graphs: For instance, for a stencil-like application, each process's local graph would consist only of the local tasks and the immediate neighbor tasks, independently of the number of processes in the distributed execution session.

### 4.7 Uintah

The Uintah computational framework was originally developed by Steven Parker, drawing on his experience in designing the SCIRun Problem Solving Environment [102]. Uintah solves complex fluid-structure interaction problems [13] using a task-graph representation of parallel computation and communication on adaptive, block-structured meshes. Uintah features the material point method (MPM) for structural mechanics, a multi-material fluid mechanics capability (ICE), and a finite-volume, large-eddy simulation code for combustion applications (ARCHES). Uintah is also composable in that external software such as linear solvers (PetSC, Hypre) or visualization components (VisIt) can be and have been added [103]. A key feature of Uintah is a clean separation between application-specific tasks and the runtime system that executes them. The application tasks for a target simulation are assembled from an XML input file specification. Each component parses input parameters from a section of an XML document while the `scheduleComputeStableTimestep` routine computes a stable timestep for the next integration interval, and the `scheduleTimeAdvance` routine schedules tasks that complete a timestep integration. Application components delegate decisions about parallelism to a scheduler component. As a global task-graph is unlikely to scale, Uintah uses a distributed tensor product task-graph, in which each node contains only the tasks it owns and has the implicit ability to communicate with tasks on other nodes. For each task-graph, there is a Data Warehouse containing the data required by the tasks. The DataWarehouse abstraction is sufficiently high-level that it can be efficiently mapped onto both message-passing and shared-memory communication mechanisms, including Kokkos [104].

The original Uintah framework (1997-2008) used static task-graph execution. When static execution became a barrier to scalability due to MPI_wait times growing on large runs, Parker's original vision was expanded to use asynchronous task execution [105, 106] and block-structured adaptive mesh refinement (BSAMR) [107]. This tiled BSAMR mesh refinement algorithm is an embarrassingly parallel method with linear complexity. Asynchronous task execution made it possible to choose alternate tasks to help avoid communication delays by overlapping computation with communication. This approach allowed Uintah to achieve both weak and strong scalability on three of the seven fastest computers in 2012, DOEs Titan and Mira and NSFs Stampede, and on almost every "leadership class" DOE parallel computer since. Uintah relies on task parallelism of the computation exceeding the physical parallelism available. Furthermore, even if each task-graph is linear, there is the potential to hide delays by switching to another task-graph.

Although Uintah worked well with PDE stencil methods with local communication, thermal radiation, the dominant mode of heat transfer in next-generation clean coal boilers [108], proved to be challenging due to its all-to-all physical (and computational) connectivity. The scaling of these calculations in the Uintah code on Titan to 16,384 GPUs was achieved through a combination of 1) reverse Monte Carlo ray

| | GPU support | | | | Others | | Sections | |
|---|---|---|---|---|---|---|---|---|
| AMT | AMD | Nvidia | Intel | Perf Port | Communication | Memory | AMT | App |
| Chapel | | | ✗ | LLVM | libfabric, GASNet, MPI, UDP, uGNI | PGAS | 4.1 | 6.1 |
| Charm++ | | | ✗ | Kokkos | LCI, UDP, MPI, libfabric, UCX, uGNI, PAMI | Actor model | 4.2 | 6.2 |
| HPX | | | | Kokkos | MPI, LCI/libfabric, GasNet, OpenSHMEM | AGAS | 4.3 | 6.3 |
| Legion | | | ~ | Kokkos, LLVM | GASNet | PGAS | 4.4 | 6.4 |
| PaRSEC | | | ~ | Agnostic | MPI, (LCI) | DMS, Dataflow | 4.5 | 6.5 |
| StarPU | | | | OpenCL | MPI | DMS | 4.6 | 6.6 |
| Uintah | | | | Kokkos | MPI | DMS | 4.7 | 6.7 |

Table 1. Technical aspects of AMTs. Support of GPUs, performance portability (abstraction to avoid written raw GPU kernels, whether provided by the AMT or the AMT is agnostic to it), communication libraries, and memory models (partitioned global address space (PGAS), active global address space (AGAS), and distributed memory system (DMS)) for all surveyed AMTs. For GPU support, the following classification is used: ✓full support, ~ partial or experimental support, and ✗ no support at the time the paper was written.



tracing (RMCRT) techniques combined with AMR to reduce the amount of global communication and 2) a novel lock- and contention-free, thread-scalable data warehouse for managing MPI communications.

In moving Uintah as a whole to heterogeneous, GPU-based systems, the Kokkos performance portability layer was used to create a GPU-portable code. This GPU requirement needed a new, heterogeneous, MPI+Kokkos task scheduler which, when combined with a challenging benchmark involving CFD loops, Hypre and RMCRT, allowed us to achieve good strong scaling on most of the DOE Summit and NSF Frontera systems [104]. The initial heterogeneous task scheduler, which used raw CUDA in scheduling infrastructure, was later updated to use Kokkos equivalents, resulting in a single code that could be run unchanged on Nvidia, Intel and AMD GPUs [109]. With the resulting scheduler, an RMCRT benchmark calculation was successfully run across up to 9,216 nodes and 10,240 nodes on the DOE Frontier and Aurora systems, respectively. These experiments demonstrate the remarkable ability of Uintah to produce scalable results at extreme scale and to provide important indications about the path to scalability on post-exascale architectures [110, 111].

### 4.8 Summary

Table 1 provides a summary of technical aspects, such as GPU support, communication libraries, and memory models, across the evaluated AMT systems. The table first assesses GPU compatibility with AMD, NVIDIA, and Intel devices. Performance portability indicates if GPU-agnostic code can be written. For communication, all libraries supported by each AMT are listed. In most cases, these libraries are used via a runtime abstraction layer, meaning that no direct interaction with their native APIs is required by the application developer. Lastly, the memory model is listed. Table 2 indicates if the parallelism or functionality with respect to the terminology in Section 3 is provided by the AMT.

| | Execution Domains | Memory Management | Parame-terized | Depen-dencies | Futures | Parallel Alg | Parallel Comp | Load Balancing |
|---|---|---|---|---|---|---|---|---|
| Chapel | Application | Process & GPU | | | | | | CPU |
| Charm++ | Application | Process | | | | | | Process |
| HPX | Application, CPU | Process | | | | | | CPU |
| Legion | CPU, GPU | Process, GPU | | | | | | Process, CPU, GPU |
| PaRSEC | CPU, GPU | Process, GPU | | | | | | CPU, GPU |
| StarPU | CPU, GPU | Process, GPU | | | | | | Process CPUs+GPUs |
| Uintah | CPU, GPU | Process, GPU | | | | | | Process CPU, GPU |

Table 2. Conceptual aspects of AMTs. *Execution Domains*: the runtime may assign a *CPU* and/or *GPU* to a task, or leave the selection to the *Application*. *Memory Management*: the runtime manages the transfer of data between *Process* and may manage memory allocations on and data transfer between *GPU*. *Load Balancing*: the runtime balances the load work across *CPU* cores, across *GPUs* within a process, or *Inter-Process* (i.e., distributed load balancing). The other terms are as described in Section 3.



## 5 Challenges and Opportunities

MPI+X dominance. While AMTs are designed to address potential weaknesses in MPI+X software solutions (Section 1), a major challenge for AMTs is that the vast majority of codes running on HPC architectures still adopt the MPI+X approach [112, 113]. There are many reasons for this, starting with the relative simplicity of MPI+X and the low entry bar for computational scientists with non-CS application backgrounds. However, around 80% of the open-source HPC codes surveyed in [114] use the MPI 2 standard and do not make use of modern MPI features. More complex asynchronous backgrounds may be unattractive if performance may be obtained using MPI+X with explicitly managed communications and synchronization. The capability of the MPI+X model is paradoxically illustrated by the AMT codes, which are built upon MPI+X, albeit using MPI in asynchronous mode and hiding MPI from the application programmer. At the same time, as applications require more sophisticated solvers with unpredictable communication patterns or run on ever larger node counts, the automated and adaptive nature of AMTs prevails in continued scaling when simpler fixed MPI+X models fail to deliver [72, 80]. It is for these reasons that AMTs do particularly well in dealing with complex applications on large node counts.

A compelling counterpoint to the dominance of MPI+X in HPC is the widespread adoption of task-based programming models in data analytics and machine learning. Task-based approaches allow them to write code in a familiar, sequential style, while parallelism is automatically extracted and managed by AMTs. Some examples are: MapReduce [42], Ray [43], Spark [44], PyTorch [45], and Jax [46].

Debugging and profiling. The advantages of adaptive execution models used by AMTs also provide challenges when debugging as in order to maximize performance, execution patterns may vary with network loads and node counts. Debugging is helped by continuously improving parallel debuggers and even more so by the provision of multiple layers of trace information generated by the AMTs. However, some AMTs use their own thread manager on top of the OS threads. In this case, parallel debuggers do

not work well since they are not agnostic of the AMT's thread manager and do not provide all details. Consequently, some AMTs provide special profilers [115].

Interplay with non-AMT libraries. Given the dominance of MPI+X, the majority of HPC libraries—such as solvers, I/O frameworks, and visualization tools—are built around this paradigm. Given that most of the surveyed AMT runtimes are implemented in C/C++ or offer C/C++ bindings, integrating these libraries is generally feasible. However, challenges arise when these libraries rely on MPI or OpenMP for parallelism, as they utilize OS threads, which can conflict with the thread management system of the AMT runtime. While there are solutions [103] to this problem in AMTs, this may lead to contention between the OS thread scheduler and the AMT's internal thread manager, potentially degrading application performance. To address this issue, the nOS-V runtime system [32]—developed by the Barcelona Supercomputing Center—offers a low-level tasking runtime that supports cooperative co-execution. This enables more efficient integration with higher-level programming models by allowing shared control over computational resources.

Community building. The AMT community remains relatively small and is primarily concentrated in North America and Europe, with a smaller contingent based in Japan. In recent years, research papers and presentations on AMT runtimes have been featured at various conferences and workshops. However, until recently, there was no dedicated venue exclusively focused on AMT systems. The establishment of the "Workshop on Asynchronous Many-Task Runtime Systems and Applications" (WAMTA)[6]—inaugurated in 2023 and now held annually—marks a significant step toward fostering a cohesive community and providing a dedicated platform for collaboration and knowledge exchange. Another workshop, not solely dedicated to AMTs, is the "Annual Parallel Applications Workshop: Alternatives to MPI+X" (PAW-ATM)[7], held in conjunction with the SC conference. Another effort was the Task-Based Algorithms and Applications (TBAA) panel at SC20 [116].

Ecosystem and continuity. Most of the surveyed AMTs are developed and maintained by small teams based at universities or national laboratories. Among them, only Chapel and Legion benefit from industrial backing—by HPE and NVIDIA, respectively. Following the "golden era" of the early 2000s, securing funding for fundamental AMT research has become increasingly challenging, likely due to the proliferation of established runtimes in the field. As a result, the development and long-term sustainability of many AMTs now depends on small groups—often composed of graduate students, postdoctoral researchers, and/or a few faculty members. This situation gives rise to the so-called "bus problem": if a key contributor graduates, leaves academia, or retires, development may stall, and the project risks stagnation or abandonment [117]. Further, this problem poses challenges related to adopter buy-in. It is not uncommon for potential adopters to seek signs of long-term commitment to a project before choosing to develop on top of an AMT system. Systems impacted by the "bus problem" may be less attractive to potential adopters and difficult to justify adopting.

Complexity creep. Increasing amounts of functionality and complexity in underlying technologies, like C++, poses challenges relating to the complexity of AMT implementations. As more functionality is introduced, it is not necessarily the case that new functionality need be adopted by a given AMT. It is important to carefully consider when and how to add new functionality to an AMT. Rather than allowing

[6]https://wamta26.stellar-group.org/
[7]https://sourceryinstitute.github.io/PAW/

the underlying technology to drive AMT development, it could be advantageous to allow application-specific needs to drive AMT development. Such a practice would help ensure that the AMT better meets the needs of the target application's science domain.

Think outside the box and education. Asynchronous programming is inherently challenging for most humans, as it deviates from our natural, sequential mode of thinking. In AMT systems, all compute kernels and I/O operations must be explicitly defined as tasks. These tasks are represented differently depending on the AMT runtime, adding another layer of abstraction and complexity.

To achieve good performance, task granularity must be carefully balanced as tasks that are too small incur high overhead, while tasks that are too large limit concurrency. Additionally, to enable effective overlap of communication and computation, developers must describe precise task dependencies. This requires a non-traditional mindset, as it often involves rethinking program structure and control flow beyond conventional bulk-synchronous paradigms.

What are often perceived as "bugs" in AMT-based applications are not necessarily flaws in the runtime system, but rather cognitive mismatches-anomalies in the user's mental model of task behavior and scheduling. Therefore, successful use of AMTs demands not only a different problem decomposition strategy, but also a shift in algorithmic thinking, embracing concurrency, latency hiding, and dynamic execution.

HPC is often treated as a peripheral or "orphan" topic within the computer science curriculum. As a result, some CS students are only introduced to the standard MPI+X programming model—where X typically refers to OpenMP or CUDA—during limited course exposure or elective modules [118, 119] and others not at all. This MPI+X approach emphasizes bulk synchronous parallelism, which may not fully prepare students for the diverse range of HPC paradigms encountered in practice. Furthermore, students frequently gain hands-on experience with these paradigms only during internships or thesis projects. Consequently, the pipeline produces relatively few graduates with prior AMT programming experience, leading to a workforce that often requires significant on-the-job training. This is particularly true for multidisciplinary students in computational science.

Support for alternative accelerator architectures. Most of the surveyed systems provide GPU support across major vendors, including AMD, NVIDIA, and Intel (Table 1). However, support for other architectures, like Apple Silicon, remains limited. In parallel, specialized AI accelerators have recently emerged to address the demands of AI training and inference workloads. Since AMT runtimes are not yet widely adopted in AI workloads, support for these accelerators remains minimal.

A similar situation exists for FPGAs and DSPs, which are still largely unsupported across AMT implementations. These gaps do not reflect fundamental limitations, but rather practical constraints—such as small development teams, limited user demand, or lack of targeted funding. Thus, while support for these platforms is technically feasible, it has not yet been prioritized or implemented.

Summary. Figure 7 shows the SWOT matrix [120] for AMTs with the strengths, weaknesses, opportunities, and threats extracted by the SWOT decision-making method.

## 6 Applications

Applications using the following AMTs are surveyed: 6.1 Chapel, 6.2 Charm++, 6.3 HPX, 6.4 Legion, 6.5 ParSec, 6.6 StarPU, and 6.7 Uintah.

### 6.1 Chapel

**SWOT matrix for AMTs: Challenges and Opportunities**

| Strengths (Benefits) | Weaknesses (Challenges) |
|---|---|
| • Can enable better scaling than *MPI+X* for irregular and unbalanced applications<br>• Task-based approaches already proven successful in some applications<br>• WAMTA workshop and TBAA panel foster a growing AMT community | • Dominance of *MPI+X* makes adoption difficult<br>• Debugging and profiling is complex due to adaptive execution models<br>• Integration issues with non-AMT libraries (*e.g.*, MPI/OpenMP)<br>• AMT programming is inherently challenging, requiring new thinking<br>• HPC treated as peripheral in CS education; limited AMT-trained workforce |
| **Opportunities (Benefits)** | **Threats (Challenges)** |
| • Long-term sustainability if funding and industry backing increase<br>• Potential for broader adoption through community growth<br>• Support for diverse accelerator architectures (GPU, FPGA, AI chips) | • Small development teams vulnerable to the "bus problem"<br>• Funding challenges risk stagnation or abandonment<br>• Resistance from established *MPI+X* user base<br>• Limited support for Apple Silicon, FPGAs, DSPs, and new accelerators |

Fig. 7. SWOT matrix [120] with strengths, weaknesses, opportunities, and threats reflecting the challenges and opportunities of AMTs.

Chapel simplifies parallel programming while maintaining high performance, enabling scientists in fields as varied as astrophysics, CFD, oceanography, and quantum physics to write scalable scientific applications with a lower barrier to HPC adoption. One of its most notable applications is Arkouda [121], a NumPy-/Pandas-like library designed for massive-scale data analytics.

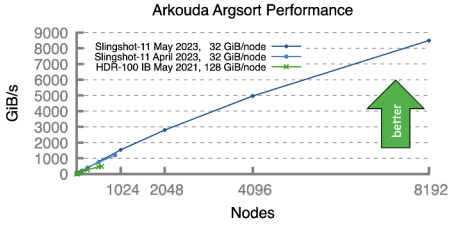


Fig. 8. Arkouda's argsort performance on HPE Cray Supercomputing EX (Slingshot 11) and HPE Apollo (Infiniband)

Arkouda bridges the gap between data science workflows and HPC systems, providing a familiar Python front-end to data scientists, backed by a Chapel server for distributed computing. Its modular design allows developers to integrate new data sources, algorithms, and workflows, making it a versatile platform for a wide range of applications, illustrated by Arachne [122], which adds graph analytics capabilities on top of Arkouda. A key objective of Arkouda is to facilitate exploratory and interactive data analysis on massive datasets. To achieve that, the server stores data in distributed memory on the HPC system.

The client communicates with the server using ZeroMQ messages [123] to perform tasks such as data manipulation and processing. The server responds with results, including small amounts of data, which can be processed locally by the client, often eliminating the need for downsampling data that can lead to missing key insights.

Arkouda has a highly performant sort [124] which can scale to thousands of compute nodes and hundreds of terabytes of data written in just ~100 lines of Chapel code, see Figure 8. This efficient sorting capability is critical in data science as it enables fast `groupby` operations, which form the basis of a lot of data science workflows.

Arkouda's sort uses Chapel's aggregation features [125] to minimize fine-grained communication, which is often a suboptimal way of communicating on networks. The aggregators combine remote copies and execute them asynchronously once their buffer is full. Additionally, Arkouda introduces automatic checkpointing, which asynchronously saves the server state, allowing users to save their work and resume later.

Another application that benefits from Chapel's asynchronous parallelism features is ChOp (Chapel-based Optimization) [126]. ChOp is a library designed to solve branch-and-bound optimization problems in distributed memory. As a combinatorial optimization problem, branch-and-bound is known to be prone to load imbalance, requiring significant engineering to be able to scale to large problem sizes. Meanwhile, the load imbalance becomes an even bigger challenge in a distributed memory setting where multiple compute nodes have to synchronize and share the work as efficiently as possible. ChOp uses Chapel's `DynamicIters` and `DistributedIters` modules to load-balance in distributed settings. These modules provide several parallel iterators, implemented at the user-level via Chapel's user-defined iterator interface [127], where the latter can specifically be used to distribute an iteration space across multiple nodes. Orthogonally, ChOp can also utilize GPUs via Chapel's native vendor-neutral GPU programming support [60]. Overall, ChOp uses Chapel's `DistributedIters` to dynamically distribute workloads across nodes. Then, within a node, the load is divided between the CPU and the GPU using one of Chapel's asynchronous task constructs, `cobegin`. Finally, while the CPU workload is parallelized dynamically with the help of `DynamicIters`, multiple GPUs within the node are used by statically dividing the work and spawning a task per GPU using a `coforall` loop. Combining Chapel's asynchronous tasks, load balancing iterators, and other parallelism and locality constructs, ChOp has been shown to scale up to 1024 AMD MI250X GPUs and 512 NVIDIA A100 GPUs on Frontier and Perlmutter respectively [126].

For additional examples of Chapel in real-world applications, see the 7 Questions for Chapel Users blog series [128].

### 6.2 Charm++

Charm++ was co-designed with multiple applications as an explicit strategy [129], and many features in Charm++ are the result of fulfilling the needs of these applications. The precursor to Charm++, the chare kernel, was developed as a runtime system to implement the Reduce-Or Process Model for parallel logic programs [130]. The Reduce and Or "processes" were essentially suspendable tasks. Each typically created other such processes and awaited responses from them, combining responses to form results to send to their parent "processes". These "processes" were implemented as activation record-like data structures called chares, where responses from children were implemented as remotely invokable functions, called entry methods. This design sufficed for newer but related classes of applications in divide-and-conquer parallelism and combinatorial search [131].

In the early 1990's, science and engineering applications became important. A CFD code, and a much larger biomolecular simulation code (NAMD) drove the next sets of abstractions. As entities in physical simulations, chares encoded iterative behavior of internal tasks with dependencies on incoming data and local task completions. This led to the development of the structured dagger notation [62] that weaves entry methods into DAGs within individual chares. Spatial over-decomposition of the simulation space in these applications led to the idea of indexed collections of chares [6], so you could easily address messages


Fig. 9. NAMD Timeline of a small subset of processors in a run on 527 nodes (~16,000 cores) on IBM© Power7-based PERC machine.

to a specific member of a collection. Unlike transient chares in tree-structured applications, chares in these applications were long lived; this necessitated the ability to migrate existing chares, including location management and measurement-based load balancing strategies.

NAMD was successful and widely adopted for biomolecular simulations that needed to scale on the largest supercomputers. It won a Gordon Bell award in 2002 for a simulation scaling to 3,000 processors of ASCI Red supercomputer. It has demonstrated scaling, e.g., to 224,000 cores of Cray XK6 [132]. It has been ported and used on almost all US supercomputers. At one point, a survey of major US academic supercomputers indicated that about $10-15\%$ of their supercomputing cycles were used for NAMD. NAMD's capability and utility was demonstrated in its use to simulate the Coronavirus responsible for the Covid-19 epidemic, which led to another Gordon Bell Award [133]. From the AMT perspective of this paper, NAMD achieved high strong scaling, using a huge number of fine-grained tasks orchestrated by the Charm++ RTS. As an example, Figure 9 shows timelines from simulation of a 92,000 atom system. Note that the time steps (indicated by the red boxes) are not synchronized globally and that many tens of tasks are scheduled on each core, based on their data dependencies, all within the small time interval of 230 $\mu$s.

ChaNGa is another Charm++ application in computational astronomy, a particle-based code for gravity and SPH calculations. Charm++ supported it with sophisticated load balancing strategies, management of fine-grained request-response pairs for tree-nodes, communication imbalances, and prioritization among tasks. It has demonstrated excellent performance, for example, scaling efficiently to about 500,000 Blue Waters cores for relatively uniform datasets and 128k cores for highly clustered datasets [134]. Figure 10 shows the time-profile, where each time interval shows a stacked chart of the activities added across all 8k cores simulating a

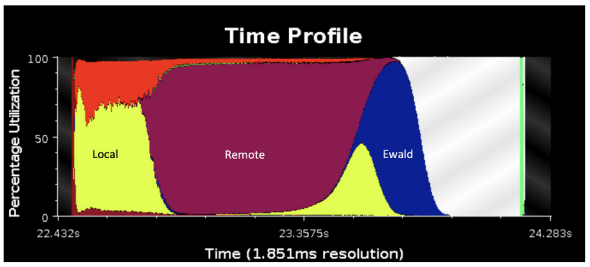


Fig. 10. ChaNGa time profile showing activities over time on a 8K core run on Blue waters, taken from [134] .

highly clustered dataset. Using priorities for tasks
allows Charm++ to run local tasks but switch to critical-path-sensitive tasks with remote dependencies as soon as they become available.

Many other applications have been developed [135] using Charm++ that are in regular use on supercomputers, including SpECTRE for multiscale simulations in gravitational physics, the Enzo-P framework (astronomy/cosmology), Quinoa (Fluid Dynamics), and OpenAtom (Quantum Chemistry for ground state and excited state).

### 6.3 HPX

A major application that has been built from the ground up using HPX is Octo-Tiger. Octo-Tiger is a grid-based astrophysics application used for the simulation of binary star mergers [136]. For example, recently it has been used to investigate the origin of R Coronae Borealis stars as the result of such mergers [137]. In Octo-Tiger, these binary star systems are modeled as self-gravitating astrophysical fluids, which requires two interleaved hydrodynamics and gravity solvers. These solvers operate on an adaptively refined grid that is implemented using an octree. Still, even with adaptive mesh refinement, Octo-Tiger has to rely on distributed runs on supercomputers to meet the computational and memory requirements posed by the intended production scenarios.

Achieving a scalable implementation is usually a daunting prospect for an application such as Octo-Tiger when faced with multiple interleaved solvers and a continuously, adaptively refined data-structure. For instance, the adaptive refinement might lead to load-balancing issues, the limited parallelism during distributed tree-traversals may lead to poor scalability and low workload per individual tree-node might lead to the starvation of the compute devices. Octo-Tiger addresses these challenges by leveraging the features offered by HPX. Its task-based approach simplifies exploiting the available parallelism during the distributed tree traversals, as the tree-structure can be naturally mapped to a task-graph. HPX's distributed features also ease the burden for load-balancing and communication, as we can call functions the same way on local and remote objects, simplifying the implementation of the tree-traversals. Furthermore, HPX's twofold integration with Kokkos (see Section 4.3) and recent developments enabling dynamic GPU kernel fusion with HPX [138] paved the road for portable and efficient GPU implementations of tree-based codes such as Octo-Tiger.

By leveraging the task-based and distributed features HPX offers and combining them with the performance portability Kokkos provides, we are able to use a wide-range of CPU and GPU supercomputers with Octo-Tiger. We recently conducted several large-scale runs with Octo-Tiger, where we showed scalability on a full system run on Perlmutter [72]. Moreover, highlighting the portability offered by Kokkos and HPX, we were also able to show Octo-Tiger's scalability on Frontier and are currently collecting initial results on Aurora, covering the major DOE supercomputers. We further showed scalability on the Supercomputer Fugaku.

Although Octo-Tiger has emerged as the flagship HPX application that showcases its capabilities at large scale, there are ongoing efforts to apply HPX to a broader range of scientific computing and machine learning domains. Notable examples include GPRat, HPX-FFT, and the HPX backend for FleCSI [139].

GPRat is a Gaussian process regression (GPR) library that leverages HPX for portability and scalability [140]. GPR occurs in various fields ranging from Kriging in geostatistics to non-linear system identification tasks in control theory. Exact GPR requires solving a dense system of linear equations. Due to its triangular structure, the applied Cholesky solver suffers from work starvation when using conventional approaches. In contrast, HPXs dataflow concept naturally captures data dependencies and ensures fully asynchronous execution. Using HPX, GPRat can be deployed in a variety of established

and emerging computing architectures. Combined with a high-level Python API, GPRat provides an accessible, portable, and scalable tool for exact GPR. Current research focuses on extending GPRats capabilities to distributed computing and integrating portable accelerator support.

HPX-FFT addresses the performance challenges associated with multidimensional large-scale Fourier transforms [141]. In distributed FFT implementations, communication is the main bottleneck. HPX mitigates this limitation by utilizing asynchronous collectives that work hand in hand with asynchronous computation tasks. A core advantage of this approach is the HPX parcelport communication layer, which enables HPX-FFT to use different communication backends.

### 6.4 Legion

To date Legion has been used for three distinct classes of applications: simulations and data analysis for science, high performance libraries used as components of other applications, and higher-level programming tools. We briefly survey examples of all three classes.

The original Legion applications were scientific simulations, with a port of S3D to Legion, an explicit combustion simulation, being the original flagship Legion code [142, 143]. Over a decade on, S3D-Legion is still used in production, continues to be updated with new features and algorithms, and has been run at scale on almost all of the top supercomputers in recent years. Much more recently, HTR, the Hypersonic Task-based Research solver, is another large chemistry simulation code built on Legion; while HTR has general capabilities including simulating combustion and turbulent flow, it has so far been used primarily to study the ignition of rockets in the vacuum of space [144, 145]. On the data analysis side, SpiniFel is a code that reconstructs molecular structure (possibly with multiple conformations) using data generated by single-particle imaging experiments [146].

Legion's expressive data partitioning and mapping have made it possible to write domain-specific languages that compile specifications into high-performance library code. Legion's dynamic nature then generally makes it straightforward to integrate the Legion code with a larger system that uses the library. We have already mentioned FlexFlow, which is a deep learning compiler built on Legion [147]. When FlexFlow was developed, the dominant strategies for distributed training were data parallelism and model parallelism. FlexFlow's innovation was to leverage Legion's data partitioning to allow every operator in a model to have its own partitioning strategy—for example some operators might be data parallel, others could instead partition the weights across GPUs, use model parallelism, or any combination of all three. A search procedure, guided by a cost model, tried many per-operator partitioning strategies to find one that was highly performant, and finally FlexFlow embedded the best parallelization strategy found in a Legion program. Because the choice of data partitioning strategy often greatly affects communication costs, finding a better data partitioning could result in dramatic speedups. Many deep learning compilers have been built in the wake of FlexFlow, exploring additional dimensions for parallelization and different search strategies (e.g., Alpa [148] and Unity [149], a direct successor to FlexFlow, among others). Another important library built on Legion is DISTAL, the DIStributed Tensor ALgebra Compiler [150]. Given a tensor algebra expression, a data partitioning strategy, a description of the machine to be targeted and a kernel schedule, DISTAL combines Legion and TACO [151] (for generating optimized kernels) to produce a specialized implementation of that specific tensor algebra computation as a Legion library. Importantly, the total specification for all four components (target expression, data partitioning, machine description, and schedule) is typically under 20 lines of code. DISTAL and the sparse version spDISTAL [152] are used commercially and code generated by these libraries has been incorporated into Legate libraries (see below).

The third class of Legion applications is higher-level programming systems. We have already mentioned Regent [153, 154], a programming language with syntactic and typechecking support for the core concepts of the programming model with the aim of providing a safer and more concise framework for writing programs compared to Legion, which is a C++ library. Interestingly, users show no consistent preference for the Legion C++ API or the Regent programming language. Some users have begun in C++ and switched to Regent because it is more productive for developers; S3D is an example. Other users have begun in Regent and switched to C++ because the C++ API is less restrictive than Regent and eliminates a dependency; HTR is an example. Pygion provides Regent-like support for the Legion programming model within Python [155] and is used for scientific projects where the users prefer a Python interface, such as SpiniFel (discussed above). FleCSI is a framework developed at Los Alamos National Laboratory for the construction of multiphysics simulations on both structured and unstructured meshes. In addition to targeting Legion as one of its backends, many of the key abstractions in the FleCSI programming model were directly inspired by Legion concepts [139]. A more recent project is NVIDIA's Legate [156], which provides a Python interface to Legion and also adds a layer that automates data partitioning and mapping, allowing users with no parallel programming experience at all to use Python to write distributed Legion programs. Legate is also designed to support the Python model of using multiple, interacting libraries. cuPyNumeric is a Legate library that parallelizes NumPy programs, and there are Legate libraries for the sparse subset of SciPy (with many of the key kernels generated by spDISTAL) and Pandas dataframes.

## 6.5 PaRSEC

PaRSEC started as a runtime for the DPLASMA [93] linear algebra library that provides dense, distributed, accelerated linear algebra routines to solve large systems of equations. Among the provided algorithms are the Cholesky, QR, and LU factorizations, Cholesky inversion, and Eigenvalue decomposition using a bulge-chasing algorithm [157]. The DPLASMA library employs the PaRSEC PTG representation, redistributable datasets [158], as well as recursive decomposition [159] to expose parallelism to the PaRSEC runtime. Key routines are GPU enabled [160–162].

PaRSEC has subsequently been used in other contexts. LibChemist is part of the NWChemEx ECP application; it consists of a large collection of computational tools for chemistry, among them a set of tensor operations implemented over TiledArray [163], within the Massively Parallel Quantum Chemistry framework (MPQC, see [164]). TiledArray has been augmented to delegate tensor contractions to the PTG DSL [157]. Initially, this migration targeted the dense scenario [165], followed by adaptation for the block-sparse situation [166].

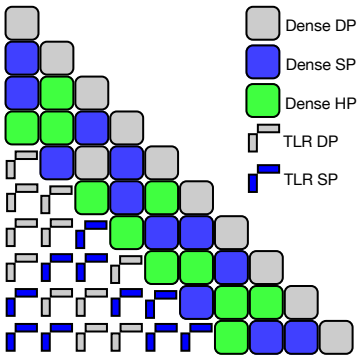


Fig. 11. Mixed precision Tile-low rank matrices in HiCMA [167].

PaRSEC has been used to port the HiCMA library to provide scalable sparse tile-low rank factorization of matrices (Figure 11) used in various scientific fields, including climate science [168] and genomics [169]. Its application in climate science has won the 2024 Gordon Bell award for climate modeling. We integrate linear algebra innovations around mixed-precision and low-rank computations into a tile-based Cholesky solver, featuring on-demand precision casting and dynamic runtime support from PaRSEC to efficiently manage tasks and data movement [167]. This adaptive methodology achieves scalability across diverse supercomputing platforms, including the Intel-based CPU system

Shaheen II, AMD-based CPU system HAWK, ARM-based CPU system Fugaku, and IBM©-based multi-GPU system Summit.

MADNESS is a framework for solving integral and differential equations using adaptive, fast methods with guaranteed precision based on multi-resolution analysis and novel separated representations [170]. MADNESS employs a continuation-passing style using distributed futures to implement operations over adaptively refined tensors representing functions in a multi-wavelet basis. This approach is being used in quantum chemistry for Hatree-Fock and density-functional theory computations, among others [171]. PaRSEC has been integrated as a task scheduler, utilizing PaRSEC's low-level infrastructure. Efficiently utilizing GPUs has been a particular challenge for MADNESS, due to the small matrices involved in representing functions in the multi-wavelet basis. Following early experiences [92], the TTG programming model is currently used to port the compute-intensive operators of MADNESS to its dataflow model, utilizing PaRSEC's robust GPU support. This work requires a rewrite of MADNESS data structures to accommodate the computational characteristics of GPUs.

### 6.6 StarPU

StarPU is the foundation AMT on which SolverStack@Inria Bordeaux [172] is built. This initiative integrates a selection of general purpose task-based numerical libraries and solvers. Chameleon [173–176] is a library that supplies routines to solve dense general systems of linear equations, symmetric positive definite systems of linear equations, and linear least squares problems, using LU, Cholesky, QR and LQ factorizations. PaStiX [177–180] is a sparse linear algebra, supernodal direct solver. Numerical algorithms are implemented in single or double precision (real or complex) using LLt, LDLt and LU with static pivoting (for non symmetric matrices having a symmetric pattern). This solver also provides some low-rank compression methods to reduce the memory footprint and/or the time-to-solution. qr_mumps [181, 182] is a sparse linear algebra, multifrontal direct solver. It implements a direct solution method based on the QR or Cholesky factorization of the input matrix. Therefore, it is suited to solving sparse least-squares problems, to computing the minimum-norm solution of sparse, underdetermined problems and to solving symmetric, positive-definite sparse linear systems. Composyx [183, 184] is a linear algebra C++ library focused on composability. Its purpose is to allow the user to express a large panel of sparse and dense linear algebra and algebraic domain decomposition algorithms using a high-level interface, for usages ranging from laptop prototypes to many nodes supercomputer parallel computations. FMR [185, 186] is a software package that supplies Fast and accurate Methods for Randomized numerical linear algebra. Cppdiodon [187] is a C++ library for Multivariate Data Analysis of large datasets. It provides methods such as Singular Value Decomposition (SVD), Principal Component Analysis (PCA), Multidimensional Scaling (MDS), and Correspondence Analysis (CoA).

The FLUSEPAő application [188, 189] is an unstructured finite-volume solver developed by Airbus Safran Launcher / ArianeGroup, in particular for simulations related to Ariane launchers. It relies on task parallelism through the StarPU task-based runtime system.

TBFMM [190] is a high-performance C++ 17 package that implements the parallel fast multipole method (FMM) in a task-based manner. TBFMM is designed to be easily customizable through C++ templating.

NNTile [191] is a task-based framework for training neural networks on heterogeneous platforms built on top of StarPU. NNTile can train the GPT2 model with 50 B parameters on a single node with 8 NVidia A100 GPUs.

### 6.7 Uintah

The development of the Uintah software with its multiphysics components (including MPM, ICE, and ARCHES) was application driven. Hence, the resulting Uintah AMT has been and can be used to solve complex, multiphysics problems involving fluids and solids.

The first scenario was the focal point of the Center for the Simulation of Accidental Fires and Explosions, a DOE Accelerated Strategic Computing Initiative center. The scenario consists of a steel container filled with explosive (PBX-9501) placed in a pool fire of JP-8 fuel [192]. Radiative heat flux from the fire heats the container and the PBX, the explosive decomposes into a gas and pressurizes the container, and the container ruptures. The gaseous products of reaction form a blast wave that expands outward along with pieces of the container and the unreacted PBX. This scenario was simulated with the original form of Uintah using ARCHES to compute the radiant fluxes to the container and the fluidstructure code (MPM/ICE) to simulate the container heating and rupture and the resulting blast wave.

The second scenario involved an actual deflagration-to-detonation-transition situation that occurred on Utah's Highway 6 when a semi-truck with $30,000$ lbs of explosives rolled over, caught fire, and detonated. To simulate this complex fluid-structure-interaction problem at the required scale, Uintah needed to scale on much larger systems using asynchronous task execution coupled with AMR to resolve the different physical scales. These improvements enabled scalable runs for the target problem and the capability to model the transition to detonation. The simulation results provided insight into the detonation's cause and a remediation strategy (changing the detonator-box packing configuration to intersperse one empty box between two full boxes in a checkerboard). Implementation of the strategy resulted in much lower simulated pressures, suggesting a safer but more expensive approach to explosives transport [193].

The third and fourth scenarios were part of the Carbon Capture Multidisciplinary Simulation Center, a DOE Predictive Science Academic Alliance Program (Phase II). The third scenario is a 1000 MWe GE Power ultra-supercritical (steam temperature $> 600$°C, net efficiency of 40%), coal-red boiler. The boiler is very large (65 m × 35 m × 15 m) with a complex geometry, including 8 corner burners firing pulverized coal (100 kg/s), 430 separated overre air inlets, and 210 miles of piping of varying metal alloys and thicknesses [194]. The boiler was simulated using ARCHES, with simulation results identifying optimization opportunities in GE's design. However, the boiler's scale necessitated a faster approach to solving the radiative transfer equations. A spatial transport sweeps method was implemented that reduced the solve times by up to a factor of 100. Further performance improvements were obtained by adding support for spatial scheduling within Uintah, since it was only necessary to schedule tasks such as radiation transport sweeps on a subset of the overall domain [194]. Other Uintah enhancements that improved simulation and data analysis efficiency included a restructuring-based parallel I/O scheme and a data parallel visualization system using OSPRay.

The final scenario, the design of a full digital twin for the 200 MW, biomass-fired tower boiler at Ontario Power's Atikokan Generating Station, extended the application-based approach to developing Uintah capability even further. In order to optimize boiler performance, a digital twin was devised that applies Bayesian inference to data from science-based models (ARCHES simulations) and from observations (Atikokan boiler) and couples it with decision theory to predict operating-variable set points that optimize boiler performance in the presence of uncertainty [195]. This digital twin moved Uintah into the domain of artificial intelligence. The development of the digital twin required data from hundreds of simulations, each requiring ~165,000 CPU-hours. The need for improved performance in such digital twin simulations has been met with the moves to the heterogeneous (CPU-GPU) runtime system as described above [109–111]. This runtime system now makes it possible to write the next generation of GPU-ready simulation codes for future simulations.

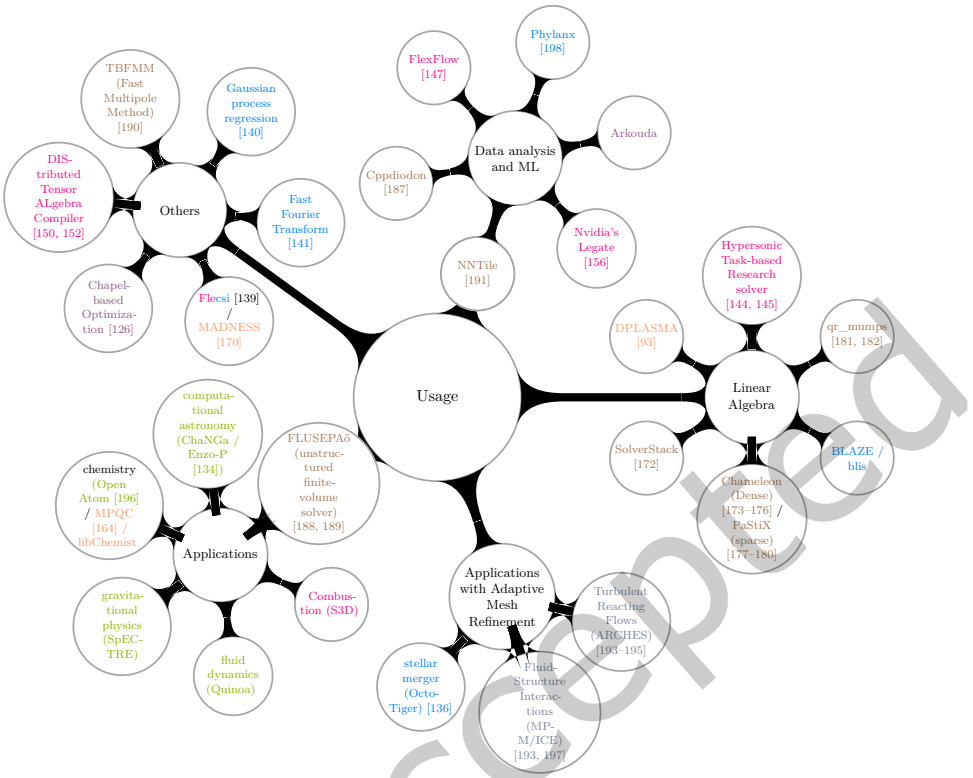


Fig. 12. Mind map of the usage of the surveyed AMTs. The color indicates the underlying AMT: Chapel, Charm++, HPX, Legion, Parsec, StarPU, and Uintah.

## 6.8 Summary

Figure 12 illustrates a mind map summarizing the aforementioned use cases. The visualization highlights three key observations: 1) StarPU and PaRSEC predominantly target linear-algebra applications; 2) AMT systems, including Charm++ and Legion, support a broader set of use cases than other frameworks; and 3) AMTs have been applied across a wide range of computational domains.

# 7 Application characteristics using AMTs

Applications that can effectively leverage AMT runtime systems typically exhibit the following characteristics:

- Irregular workloads. Such workloads arise in particle-based methods and adaptive mesh refinement. Additional representative examples are provided in Figure 12.
- Decomposability into fine-grained computational tasks. The application should be expressible as a collection of fine-grained tasks; however, tasks that are excessively small must be aggregated to achieve a balanced computation-to-overhead ratio.
- Fine-grained synchronization and data movement. The application should enable explicit synchronization and localized data exchange between tasks, while avoiding coarse-grained, bulk-synchronous execution patterns.

These characteristics address the $\mathcal{SLOW}$ aspects of most legacy applications:

- $\mathcal{S}$tarvation occurs when insufficient concurrent work is available to fully utilize all resources.
- $\mathcal{L}$atencies occur due to the delay in accessing remote resources and services.
- $\mathcal{O}$verheads are introduced for parallel execution, and these context switches introduce overheads.
- $\mathcal{W}$aiting for contention resolution due to unavailable, oversubscribed shared resources.

A nice visualization of these limitations is the woodcarving "Four Horsemen of the Apocalypse" by Albrecht Dürrer from 1511[8]. For more details, we refer to [71, Chapter 5].

## 8 Conclusion and Outlook

From a historical perspective, AMTs began to appear in the late 1980s, with only a few emerging during the 1990s. The golden era occurred in the 2000s, when the majority of AMTs were developed. Since that period, only a handful of new AMTs have emerged. It is notable that many of the golden-era AMTs are still actively being developed.

This survey indicates that AMTs provide notable benefits for adaptive and irregular applications (Section 7). The surveyed AMTs either emphasized a flagship application or presented a broader overview across multiple domains (Section 6). Use-cases included linear algebra, data analysis and machine learning, adaptive mesh refinement, molecular dynamics, computational astrophysics and chemistry, fluid dynamics, and combustion modeling (Figure 12). Nevertheless, relative to MPI+X, the integration of AMTs into scientific high-performance production codes in academic and laboratory settings remains limited. By contrast, AMTs have achieved broader adoption in software ecosystems that support machine learning and data science.

The reasons for the limited use of AMTs have been discussed in Section 5. This comprehensive survey offers readers guidance by: 1) outlining the features of the surveyed AMTs to support decisions regarding their suitability for particular applications, 2) providing insights into whether specific application types can benefit from AMTs, and 3) identifying the key challenges associated with their adoption.

May this guidance inspire others to become curious and to foster a broader adoption of AMTs in suitable applications.



## 9 Acknowledgments

This work was supported by the U.S. Department of Energy through the Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). LA-UR-25-30250. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was supported by NSF grant #1931387 Production quality Ecosystem for Programming and Executing eXtreme-scale Applications (EPEXA).

## 10 AI Use Disclosure

Generative AI tools were used in the early stages of this work to assist with summarizing background topics and compiling an initial list of relevant references. All AI-generated content was thoroughly reviewed, validated, and revised by the authors to ensure accuracy, clarity, and scholarly integrity. The final manuscript reflects the authors original analysis and intellectual contribution.

[8] https://en.wikipedia.org/wiki/Apocalypse_(D%C3%BCrer)

## References


[1] Rishiyur S Nikhil et al. 1990. Executing a program on the MIT tagged-token dataflow architecture. IEEE Transactions on computers 39, 3 (1990), 300–318.

[2] D Culler Arvind, R Iannucci, Vinod Kathail, Keshav Pingali, and R Thomas. 1984. The tagged token dataflow architecture. Distributed in subject 6 (1984).

[3] Robert M Keller, Frank CH Lin, and Jiro Tanaka. 1984. Rediflow multiprocessing. Proceedings of IEEE Compcon (1984).

[4] Balkrishna Ramkumar and Laxmikant V Kalé. 1990. A Chare kernel implementation of a parallel Prolog compiler. In Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming. 99–108.

[5] Robert D Blumofe et al. 1995. Cilk: An efficient multithreaded runtime system. ACM SigPlan Notices 30, 8 (1995), 207–216.

[6] Orion S Lawlor and Laxmikant V Kalé. 2001. Supporting dynamic parallel object arrays. In Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande. 21–28.

[7] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice. In SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC).

[8] Laxmikant V Kalé and Gengbin Zheng. 2013. The Charm++ Programming Model. (2013).

[9] Hartmut Kaiser et al. 2020. HPX - The C++ Standard Library for Parallelism and Concurrency. Journal of Open Source Software 5, 53 (2020), 2352. DOI:http://dx.doi.org/10.21105/joss.02352

[10] Bradford L. Chamberlain. 2015. Chapel. In A Brief Overview of Parallel Programming Models, Pavan Balaji (Ed.). MIT Press, Chapter 6, 129–159.

[11] Cédric Augonnet et al. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23, 2 (2011), 187–198. DOI:http://dx.doi.org/10.1002/cpe.1631

[12] Rob Pike. 2012. Go at Google: Language Design in the Service of Software Engineering. Commun. ACM 55, 2 (2012), 44–49.

[13] S.G. Parker et al. 2006. A component-based parallel infrastructure for the simulation of fluidstructure interaction. Engineering with Computers 22 (2006), 277–292.

[14] Kyle B Wheeler et al. 2008. Qthreads: An API for programming with millions of lightweight threads. In 2008 IEEE International Symposium on Parallel and Distributed Processing. IEEE, 1–8.

[15] Arch D. Robison. 2013. Composable Parallel Patterns with Intel Cilk Plus . Computing in Science & Engineering 15, 02 (March 2013), 66–71. DOI:http://dx.doi.org/10.1109/MCSE.2013.21

[16] George Bosilca et al. 2012. DAGuE: A generic distributed DAG Engine for High Performance Computing. Parallel Comput. 38, 1-2 (2012-00 2012), 27–51.

[17] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.

[18] John Bachan et al. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 963–973. DOI:http://dx.doi.org/10.1109/IPDPS.2019.00104

[19] Jeff Bezanson et al. 2017. Julia: A fresh approach to numerical computing. SIAM review 59, 1 (2017), 65–98.

[20] Message Passing Interface Forum. 2025. MPI: A Message-Passing Interface Standard.

[21] Jonathan Lifflander, Phil Miller, Nicole Lemaster Slattengren, Nicolas Morales, Paul Stickney, and Philippe P Pébaÿ. 2020. Design and implementation techniques for an MPI-oriented AMT runtime. In 2020 Workshop on Exascale MPI (ExaMPI). IEEE, 31–40.

[22] Arch D. Robison. 2011. Intelő Threading Building Blocks (TBB). Springer US, Boston, MA, 955–964. DOI:http://dx.doi.org/10.1007/978-0-387-09766-4_51

[23] 2024. OpenMP OpenMP Application Programming Interface, Version 6.0. (Nov 2024). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf

[24] Atmn Patel and Johannes Doerfert. 2022. Remote OpenMP offloading. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22). Association for Computing Machinery, New York, NY, USA, 441442. DOI:http://dx.doi.org/10.1145/3503221.3508416

[25] Baodi Shan, Mauricio Araya-Polo, and Barbara Chapman. 2025. DiOMP-Offloading: Toward Portable Distributed Heterogeneous OpenMP. In Proceedings of the SC'25 Workshops of the International Conference for High Performance

Computing, Networking, Storage and Analysis. 1289–1301.
[26] Alejandro Fernández, Vicenç Beltran, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. 2014. Task-Based Programming with OmpSs and Its Application. In Euro-Par 2014: Parallel Processing Workshops, Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander (Eds.). Springer International Publishing, Cham, 601–612.
[27] Adrian Munera, Sara Royuela, Roger Ferrer, Raul Peñacoba, and Eduardo Quiñones. 2020. Static Analysis to Enhance Programmability and Performance in OmpSs-2. In High Performance Computing, Heike Jagode, Hartwig Anzt, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 19–33.
[28] M. Cosnard and M. Loi. 1995. Automatic task graph generation techniques. In Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, Vol. 2. 113–122 vol.2. DOI: http://dx.doi.org/10.1109/HICSS.1995.375471
[29] Bérenger Bramas. 2019. Increasing the degree of parallelism using speculative execution in task-based runtime systems. PeerJ Computer Science 5 (2019), e183.
[30] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S Vetter. 2021. IRIS: A portable runtime system exploiting multiple heterogeneous programming systems. In 2021 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 1–8.
[31] Tsung-Wei Huang et al. 2021. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. IEEE TPDS (2021). DOI: http://dx.doi.org/10.1109/TPDS.2021.3104255
[32] David Álvarez et al. 2024. nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling. In 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 312–324. DOI: http://dx.doi.org/10.1109/IPDPS57955.2024.00035
[33] Peter Thoman et al. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. The Journal of Supercomputing 74, 4 (2018), 1422–1434.
[34] Patrick Diehl et al. 2023. Benchmarking the parallel 1d heat equation solver in chapel, charm++, C++, HPX, go, julia, python, rust, swift, and Java. In European Conference on Parallel Processing. Springer, 127–138.
[35] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. 2004. X10: Programming for hierarchical parallelism and non-uniform data access. In Proceedings of the International Workshop on Language Runtimes, OOPSLA, Vol. 30.
[36] Jinpil Lee and Mitsuhisa Sato. 2010. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In 2010 39th International Conference on Parallel Processing Workshops. IEEE, 413–420.
[37] Joseph Schuchart and Jose Gracia. 2019. Global Task Data-Dependencies in PGAS Applications. Springer, 312–329. DOI: http://dx.doi.org/10.1007/978-3-030-20656-7_16
[38] Shumpei Shiina and Kenjiro Taura. 2023. Itoyori: Reconciling Global Address Space and Global Fork-Join Task Parallelism. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23). Association for Computing Machinery, New York, NY, USA, Article 14, 15 pages. DOI: http://dx.doi.org/10.1145/3581784.3607049
[39] Rabab Alomairy, Felipe Tome, Julian Samaroo, and Alan Edelman. 2024. Dynamic Task Scheduling with Data Dependency Awareness Using Julia. In 2024 IEEE High Performance Extreme Computing Conference (HPEC). 1–7. DOI: http://dx.doi.org/10.1109/HPEC62836.2024.10938467
[40] Timothy Blattner, Walid Keyrouz, Mary Brady, Shuvra Bhattacharyya, and Milton Halem. 2017. A Hybrid Task Graph Scheduler for High Performance Image Processing Workflows. Journal of Signal Processing Systems (2017-06-22 2017). DOI: http://dx.doi.org/10.1007/s11265-017-1262-6
[41] Cédric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland. 2024. CUDASTF: Bridging the gap between cuda and task parallelism. In SC24: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1–17.
[42] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI). USENIX Association, 137–150.
[43] Philipp Moritz et al. 2017. Ray: A Distributed Framework for Emerging AI Applications. arXiv preprint arXiv:1712.05889 (dec 2017).
[44] Matei Zaharia et al. 2016. Apache Spark: A Unified Engine for Big Data Processing. Commun. ACM 59, 11 (2016), 56–65. DOI: http://dx.doi.org/10.1145/2934664
[45] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems, Vol. 32. Curran Associates, Inc.

[46] Roy Frostig, Matthew James Johnson, and Chris Leary. 2019. Compiling machine learning programs via high-level tracing. In SysML conference 2018.
[47] Elliott Slaughter et al. 2020. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. 1–15. DOI: http://dx.doi.org/10.1109/SC41405.2020.00066
[48] Nanmiao Wu et al. 2023. Quantifying Overheads in Charm++ and HPX Using Task Bench. In Euro-Par 2022: Parallel Processing Workshops, Jeremy Singer, Yehia Elkhatib, Dora Blanco Heras, Patrick Diehl, Nick Brown, and Aleksandar Ilic (Eds.). Springer Nature Switzerland, Cham, 5–16.
[49] Reazul Hoque and Pavel Shamis. 2018. Distributed Task-Based Runtime Systems - Current State and Micro-Benchmark Performance. In 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). 934–941. DOI: http://dx.doi.org/10.1109/HPCC/SmartCity/DSS.2018.00155
[50] George Almasi. 2011. PGAS (partitioned global address space) languages. In Encyclopedia of Parallel Computing. Springer, 1539–1545.
[51] Christian R. Trott et al. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. IEEE Transactions on Parallel and Distributed Systems 33, 4 (2022), 805–817. DOI: http://dx.doi.org/10.1109/TPDS.2021.3097283
[52] David A. Beckingsale et al. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). 71–81. DOI: http://dx.doi.org/10.1109/P3HPC49587.2019.00012
[53] Joseph Schuchart et al. 2016. The shift from processor power consumption to performance variations: fundamental implications at scale. Comput. Sci. 31, 4 (Nov. 2016), 197205. DOI: http://dx.doi.org/10.1007/s00450-016-0327-2
[54] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. 2008. Characterizing application sensitivity to OS interference using kernel-level noise injection. In SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. 1–12. DOI: http://dx.doi.org/10.1109/SC.2008.5219920
[55] David Callahan, Bradford L. Chamberlain, and Hans Zima. 2004. The Cascade High Productivity Language. In Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04). IEEE, 52–60.
[56] Chapel website. Chapel Programming Language Website. https://chapel-lang.org/. ([n. d.]). Accessed: 2025-10-03.
[57] Chapel Team. Chapel 1.31/1.32 Release Notes: SS-11 / IB Performance Status. https://chapel-lang.org/releaseNotes/1.31-1.32/04-perf-ss11-ib.pdf. ([n. d.]).
[58] Guillaume Helbecque, Jan Gmys, Tiago Carneiro, Nouredine Melab, and Pascal Bouvry. 2022. A performance-oriented comparative study of the Chapel high-productivity language to conventional programming environments. In Proceedings of the Thirteenth International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '22). Association for Computing Machinery, New York, NY, USA, 2129. DOI: http://dx.doi.org/10.1145/3528425.3529104
[59] Chapel Developer Community. 2025. Task Parallelism and Synchronization. (Sept. 2025). https://chapel-lang.org/docs/language/spec/task-parallelism-and-synchronization.html.
[60] Engin Kayraklioglu and Andy Stone. 2024. Productive, Vendor-Neutral GPU Programming Using Chapel. In SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1914–1922.
[61] Josh Milthorpe, Xianghao Wang, and Ahmad Azizi. 2024. Performance Portability of the Chapel Language on Heterogeneous Architectures. In 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). 6–13. DOI: http://dx.doi.org/10.1109/IPDPSW63119.2024.00011
[62] L. V. Kale and Milind Bhandarkar. 1996. Structured Dagger: A Coordination Language for Message-Driven Programming. In Proceedings of Second International Euro-Par Conference (Lecture Notes in Computer Science), Vol. 1123-1124. 646–653.
[63] Bilge Acun, Akhil Langer, Esteban Meneses, Harshitha Menon, Osman Sarood, Ehsan Totoni, and Laxmikant V Kalé. 2016. Power, Reliability, and Performance: One System to Rule them All. Computer 49, 10 (2016), 30–37.
[64] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. 2004. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In 2004 IEEE Cluster. San Diego, CA, 93–103.
[65] Abhishek Gupta, Paolo Faraboschi, Filippo Gioachin, Laxmikant V Kale, Richard Kaufmann, Bu-Sung Lee, Verdi March, Dejan Milojicic, and Chun Hui Suen. 2014. Evaluating and improving the performance and scheduling of HPC applications in cloud. IEEE Transactions on Cloud Computing 4, 3 (2014), 307–321.
[66] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. 2003. Scaling Molecular Dynamics to 3000 Processors with Projections: A Performance Analysis Case Study. In Terascale Performance Analysis Workshop,

International Conference on Computational Science(ICCS). Melbourne, Australia.
[67] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. 2006. Performance Evaluation of Adaptive MPI. In Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006.
[68] British Jetley. 2014. Incompleteness+ interoperability: a multi-paradigm approach to parallel programming for science and engineering applications. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[69] Juan J Galvez, Karthik Senthil, and Laxmikant Kale. 2018. Charmpy: A python parallel programming model. In 2018 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 423–433.
[70] Aditya Bhosale, Zane Fink, and Laxmikant Kale. 2024. An Abstraction for Distributed Stencil Computations Using Charm++. In Workshop on Asynchronous Many-Task Systems and Applications. Springer, 123–134.
[71] Patrick Diehl et al. 2024. Parallel C++: Efficient and Scalable High-Performance Parallel Programming Using HPX. Springer Nature.
[72] Gregor DaiSS, Patrick Diehl, Jiakun Yan, John K. Holmen, Rahulkumar Gayatri, Christoph Junghans, Alexander Straub, Jeff R. Hammond, Dominic Marcello, Miwako Tsuji, Dirk Pflüger, and Hartmut Kaiser. 0. Asynchronous-many-task systems: Challenges and opportunities - Scaling an AMR astrophysics code on exascale machines using Kokkos and HPX. The International Journal of High Performance Computing Applications 0, 0 (0), 10943420251386503. DOI: http://dx.doi.org/10.1177/10943420251386503 arXiv:https://doi.org/10.1177/10943420251386503
[73] Hartmut Kaiser et al. 2009. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In Parallel Processing Workshops. IEEE Computer Society, Los Alamitos, CA, USA, 394–401. DOI:http://dx.doi.org/10.1109/ICPPW.2009.14
[74] Gregor Daiß et al. 2021. Beyond fork-join: Integration of performance portable Kokkos kernels with HPX. In 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 377–386.
[75] Patrick Diehl et al. 2024. Simulating stellar merger using HPX/Kokkos on A64FX on Supercomputer Fugaku. The Journal of Supercomputing 80, 12 (2024), 16947–16978.
[76] Patrick Diehl et al. 2023. Evaluating HPX and Kokkos on RISC-V using an astrophysics application Octo-Tiger. In Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. 1533–1542.
[77] Patrick Diehl et al. 2024. Preparing for HPC on RISC-V: Examining Vectorization and Distributed Performance of an Astrophysics Application with HPX and Kokkos. In SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1656–1665. DOI:http://dx.doi.org/10.1109/SCW63240.2024.00207
[78] Parsa Amini and Harmut Kaiser. 2019. Assessing the Performance Impact of using an Active Global Address Space in HPX: A Case for AGAS. In 2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM). 26–33. DOI:http://dx.doi.org/10.1109/IPDRM49579.2019.00008
[79] Jiakun Yan et al. 2023. Design and Analysis of the Network Software Stack of an Asynchronous Many-task System – The LCI parcelport of HPX. In Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23). Association for Computing Machinery, New York, NY, USA, 11511161.
[80] Gregor Daiß, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pfüger. 2019. From piz daint to the stars: Simulation of stellar mergers using high-level abstractions. In Proceedings of the international conference for high performance computing, networking, storage and analysis. 1–37.
[81] Patrick Diehl et al. 2021. Performance measurements within asynchronous task-based runtime systems: A double white dwarf merger as an application. Computing in Science & Engineering 23, 3 (2021), 73–81.
[82] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. Information and computation 132, 2 (1997), 109–176.
[83] David Gay and Alex Aiken. 2001. Language support for regions. In Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation. 70–80.
[84] Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. 2023. Visibility Algorithms for Dynamic Dependence Analysis and Distributed Coherence. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 218–231.
[85] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent partitioning. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 344–358.

[86] Thiago SFX Teixeira, Alexandra Henzinger, Rohan Yadav, and Alex Aiken. 2023. Automated mapping of task-based programs onto distributed and heterogeneous machines. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1–13.

[87] Allen Wei, Alex Nie, Thiago Teixeira, Rishabh Yadav, William Lee, Kevin Wang, and Alex Aiken. 2025. Improving Parallel Program Performance with LLM Optimizers via Agent-System Interfaces. In Proceedings of the International Conference on Machine Learning.

[88] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. 2017. Control Replication: Compiling implicit parallelism to efficient SPMD with logical regions. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1–12.

[89] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. 2021. Scaling implicit parallelism via dynamic control replication. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 105118.

[90] Reazul Hoque et al. 2017. Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '17). Association for Computing Machinery, New York, NY, USA, Article 6, 8 pages. DOI:http://dx.doi.org/10.1145/3148226.3148233

[91] G. Bosilca et al. 2020. The Template Task Graph (TTG) - an emerging practical dataflow programming paradigm for scientific simulation at extreme scale. In 2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2). 1–7. DOI:http://dx.doi.org/10.1109/ESPM251964.2020.00011

[92] Joseph Schuchart et al. 2022. Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment. In 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 839–849. DOI:http://dx.doi.org/10.1109/IPDPS53621.2022.00086

[93] George Bosilca et al. 2011. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum. 1432–1441. DOI:http://dx.doi.org/10.1109/IPDPS.2011.299

[94] Omri Mor, George Bosilca, and Marc Snir. 2023. Improving the Scaling of an Asynchronous Many-Task Runtime with a Lightweight Communication Engine. In Proceedings of the 52nd International Conference on Parallel Processing (ICPP '23). Association for Computing Machinery, New York, NY, USA, 153162. DOI:http://dx.doi.org/10.1145/3605573.3605642

[95] Thomas Herault, Joseph Schuchart, Edward F. Valeev, and George Bosilca. 2022. Composition of Algorithmic Building Blocks in Template Task Graphs. In 2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM). 26–38. DOI:http://dx.doi.org/10.1109/PAW-ATM56565.2022.00008

[96] Emmanuel Agullo et al. 2016. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. Research Report RR-8927. Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA. 27 pages. https://inria.hal.science/hal-01332774

[97] Emmanuel Agullo et al. 2017. Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. IEEE Transactions on Parallel and Distributed Systems (April 2017), 14. DOI:http://dx.doi.org/10.1109/TPDS.2017.2697857

[98] Maxime Gonthier, Loris Marchal, and Samuel Thibault. 2023. Taming data locality for task scheduling under memory constraint in runtime systems. Future Generation Computer Systems 143 (2023), 305–321. DOI:http://dx.doi.org/10.1016/j.future.2023.01.024

[99] Alexandre Denis et al. 2020. Using Dynamic Broadcasts to improve Task-Based Runtime Performances. In Euro-Par 2020 (Euro-Par 2020). Rzadca and Malawski, Springer, Warsaw, Poland. DOI:http://dx.doi.org/10.1007/978-3-030-57675-2_28

[100] Romain Lion and Samuel Thibault. 2020. From tasks graphs to asynchronous distributed checkpointing with local restart. In FTXS 2020 - IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale. Atlanta / Virtual, United States. DOI:http://dx.doi.org/10.1109/FTXS51974.2020.00009

[101] Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Thomas Morin, Samuel Thibault, and Pierre-André Wacrenier. 2025. Optimizing Parallel Heterogeneous System Efficiency: Dynamic Task Graph Adaptation with Recursive Tasks. Journal of Parallel and Distributed Computing 205 (June 2025), 105157. DOI:http://dx.doi.org/10.1016/j.jpdc.2025.105157

[102] Steven G. Parker et al. 1997. The SCIRun Computational Steering Software System. Birkhäuser Boston, Boston, MA, 5–44. DOI:http://dx.doi.org/10.1007/978-1-4612-1986-6_1

[103] D. Sahasrabudhe, R. Zambre, A. Chandramowlishwaran, and M. Berzins. 2020. Optimizing the Hypre solver for manycore and GPU architectures. In Journal of Computational Science. Springer International Publishing, 101279. http://www.sci.utah.edu/publications/Sah2020b/00_main.pdf

[104] J.K. Holmen et al. 2022. Porting Uintah to Heterogeneous Systems. In Proceedings of the Platform for Advanced Scientific Computing Conference (PASC22) Best Paper Award. ACM. http://www.sci.utah.edu/publications/Hol2022b/pasc22.pdf

[105] Q. Meng, J. Luitjens, and M. Berzins. 2010. Dynamic Task Scheduling for the Uintah Framework. In Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10). 1–10. http://www.sci.utah.edu/publications/Men2010b/Meng_TaskSchedulingUintah2010.pdf

[106] Q. Meng et al. 2013. Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers. In Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis. 96:1–96:12. http://www.sci.utah.edu/publications/meng13/Meng_SC2013.pdf

[107] J. Luitjens and M. Berzins. 2011. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. Concurrency And Computation: Practice And Experience 23, 13 (2011), 1522–1537. http://www.sci.utah.edu/publications/Lui2011a/Luitjens-Berzins_CCPE2011.pdf

[108] A. Humphrey et al. 2016. Radiative Heat Transfer Calculation on 16384 GPUs Using a Reverse Monte Carlo Ray Tracing Approach with Adaptive Mesh Refinement. In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). 1222–1231. http://www.sci.utah.edu/publications/Hum2016a/ipdps-pdsec16.pdf

[109] J. K. Holmen, M. García, A. Bagusetty, A. Sanderson, and M. Berzins. 2024. Making Uintah Performance Portable for Department of Energy Exascale Testbeds. In Euro-Par 2023: Parallel Processing. 1–12.

[110] J.K. Holmen et al. 2025. Lessons Learned and Scalability Achieved when Porting Uintah to DOE Exascale Systems. In Proceedings of the AMTE workshop (accepted). http://www.sci.utah.edu/publications/Hol2025a/AMTE24-INITIAL.pdf

[111] J.K. Holmen et al. 2025. Scaling Uintah on the Aurora Exascale System up to 122,880 Intel Ponte Vecchio Xe Stacks. In Proceedings of the PEARC 2025 Conference (accepted).

[112] Kurt B. Ferreira and Scott Levy. 2021. Evaluating MPI resource usage summary statistics. Parallel Comput. 108 (2021). DOI:http://dx.doi.org/10.1016/j.parco.2021.102825 Cited by: 2; All Open Access, Bronze Open Access.

[113] Ganesh Gopalakrishnan et al. 2011. Formal analysis of MPI-based parallel programs. Commun. ACM 54, 12 (2011), 82–91.

[114] Ignacio Laguna et al. 2019. A large-scale study of MPI usage in open-source HPC applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19). Association for Computing Machinery, New York, NY, USA, Article 31, 14 pages. DOI:http://dx.doi.org/10.1145/3295500.3356176

[115] Kevin A Huck et al. 2015. An autonomic performance environment for exascale. Supercomputing frontiers and innovations 2, 3 (2015), 49–66.

[116] Bryce Adelstein-Lelbach et al. 2021. TBAA20: Task-Based Algorithms and Applications. Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States); NVIDIA Corporation, Santa Clara, CA (United States); Univ. of Illinois at Urbana-Champaign, IL (United States); Louisiana State Univ., Baton Rouge, LA (United States); Oracle Corporation, Redwood City, CA (United States); Univ. of Hawaii, Honolulu, HI (United States). DOI:http://dx.doi.org/10.2172/1764191

[117] Guilherme Avelino, Marco Tulio Valente, and Andre Hora. 2017. What is the Truck Factor of popular GitHub applications? A first assessment. Technical Report. PeerJ Preprints.

[118] Cody Stevens, Sean Anderson, and Adam Carlson. 2024. Integrating High Performance Computing into Higher Education and the Pedagogy of Cluster Computing. In Practice and Experience in Advanced Research Computing 2024: Human Powered Computing (PEARC '24). Association for Computing Machinery, New York, NY, USA, Article 106, 3 pages. DOI:http://dx.doi.org/10.1145/3626203.3670588

[119] Pawel Czarnul, Mariusz Matuszek, and Adam Krzywaniak. 2024. Teaching High–performance Computing Systems – A Case Study withăParallel Programming APIs: MPI, OpenMP andăCUDA. In Computational Science – ICCS 2024, Leonardo Franco, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer Nature Switzerland, Cham, 398–412.

[120] Paul C Nutt and Robert W Backoff. 1993. Transforming public organizations with strategic management and strategic leadership. Journal of management 19, 2 (1993), 299–347.

[121] Michael Merrill, William Reus, and Timothy Neumann. 2019. Arkouda: interactive data exploration backed by Chapel. In Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop. 28–28.

[122] Oliver Andres Alvarado Rodriguez. 2025. On the Design of a Framework for Large-Scale Exploratory Graph Analytics. Ph.D. Dissertation. New Jersey Institute of Technology, Newark, New Jersey, USA. Advisor(s) David A. Bader. https://digitalcommons.njit.edu/dissertations/1829

[123] Pieter Hintjens. 2013. ZeroMQ: messaging for many applications. " O'Reilly Media, Inc.".

[124] Brad Chamberlain. 2024. Arkouda and Chapel: Highlights Since CLSAC 2022. In CLSAC 2024. https://chapel-lang.org/presentations/ChapelForCLSAC2024-presented.pdf Presentation slides, Slide 11.

[125] Engin Kayraklioglu, Elliot Ronaghan, Michael P Ferguson, and Bradford L Chamberlain. 2021. Locality-based optimizations in the Chapel compiler. In International Workshop on Languages and Compilers for Parallel Computing. Springer, 3–17.

[126] Tiago Carneiro, Engin Kayraklioglu, Guillaume Helbecque, and Nouredine Melab. 2024. Investigating Portability in Chapel for Tree-based Optimization on GPU-powered Clusters. In European Conference on Parallel Processing. Springer, 386–399.

[127] Bradford L Chamberlain, Sung-Eun Choi, Steven J Deitz, and Angeles Navarro. 2011. User-defined parallel zippered iterators in Chapel. In Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models, Vol. 2011. 1–11.

[128] Brad Chamberlain and Engin Kayraklioglu. 2025. 7 Questions for Chapel Users. Chapel blog. (Sept. 2025). https://chapel-lang.org/blog/series/7-questions-for-chapel-users/

[129] Laxmikant V Kalé. 1994. Application oriented and computer science centered HPCC research. In Developing a computer science agenda for high-performance computing. ACM, 98–105.

[130] L.V.Kalè and B. Ramkumar. 1992. The Reduce-OR-Process Model for Parallel Logic Programming on Nonshared Memory Machines. John Wiley, 55–84. Editors: M. Wise and Peter Kacssuk.

[131] L.V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. 1993. Prioritization in Parallel Symbolic Computing. In Lecture Notes in Computer Science, T. Ito and R. Halstead (Eds.), Vol. 748. Springer-Verlag, 12–41.

[132] Yanhua Sun, Gengbin Zheng, Chao Mei Eric J. Bohm, Terry Jones, Laxmikant V. Kalé, and James C.Phillips. 2012. Optimizing Fine-grained Communication in a Biomolecular Simulation Application on Cray XK6. In Proceedings of the 2012 ACM/IEEE conference on Supercomputing. Salt Lake City, Utah.

[133] Lorenzo Casalino et al. 2021. AI-driven multiscale simulations illuminate mechanisms of SARS-CoV-2 spike dynamics. The International Journal of High Performance Computing Applications 35, 5 (2021), 432–451.

[134] Harshitha Menon, Lukasz Wesolowski, Gengbin Zheng, Pritish Jetley, Laxmikant Kale, Thomas Quinn, and Fabio Governato. 2015. Adaptive techniques for clustered N-body cosmological simulations. Computational Astrophysics and Cosmology 2, 1 (2015), 1.

[135] Laxmikant V Kale and Abhinav Bhatele. 2014. Parallel science and engineering applications: The Charm++ approach. CRC Press.

[136] Dominic C Marcello et al. 2021. Octo-Tiger: a new, 3D hydrodynamic code for stellar mergers that uses HPX parallelization. Monthly Notices of the Royal Astronomical Society 504, 4 (04 2021), 5345–5382. DOI: http://dx.doi.org/10.1093/mnras/stab937 arXiv:https://academic.oup.com/mnras/article-pdf/504/4/5345/37975469/stab937.pdf

[137] Sagiv Shiber et al. 2024. Hydrodynamic simulations of white dwarfwhite dwarf mergers and the origin of R Coronae Borealis stars. Monthly Notices of the Royal Astronomical Society 535, 2 (10 2024), 1914–1943. DOI: http://dx.doi.org/10.1093/mnras/stae2343

[138] Gregor DaiSS, Patrick Diehl, Dominic Marcello, Alireza Kheirkhahan, Hartmut Kaiser, and Dirk Pflüger. 2022. From Task-Based GPU Work Aggregation to Stellar Mergers: Turning Fine-Grained CPU Tasks into Portable GPU Kernels. In 2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). 89–99. DOI: http://dx.doi.org/10.1109/P3HPC56579.2022.00014

[139] Ben Bergen, Irina Demeshko, Charles Ferenbaugh, Davis Herring, Li-Ta Lo, Julien Loiseau, Navamita Ray, and Andrew Reisner. 2021. FleCSI 2.0: The Flexible Computational Science Infrastructure Project. In European Conference on Parallel Processing. Springer, 480–495.

[140] Maksim Helmann, Alexander Strack, and Dirk Pflüger. 2026. GPRat: Gaussian Process Regression withăAsynchronous Tasks. In Asynchronous Many-Task Systems and Applications, Patrick Diehl, Qinglei Cao, Thomas Herault, and George Bosilca (Eds.). Springer Nature Switzerland, Cham, 83–94.

[141] Alexander Strack, Christopher Taylor, Patrick Diehl, and Dirk Pflüger. 2024. Experiences Porting Shared andăDistributed Applications toăAsynchronous Tasks: A Multidimensional FFT Case-Study. In Asynchronous Many-Task Systems and Applications, Patrick Diehl, Joseph Schuchart, Pedro Valero-Lara, and George Bosilca (Eds.). Springer Nature Switzerland, Cham, 111–122.

[142] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2014. Structure slicing: Extending logical regions with fields. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 845–856.

[143] Sean Treichler, Michael Bauer, Ankit Bhagatwala, Giulio Borghesi, Ramanan Sankaran, Hemanth Kolla, Patrick S McCormick, Elliott Slaughter, Wonchan Lee, Alex Aiken, et al. 2017. S3d-legion: An exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry. In Exascale Scientific Applications. Chapman and Hall/CRC, 257–278.

[144] Mario Di Renzo, Lin Fu, and Javier Urzay. 2020. HTR solver: An open-source exascale-oriented task-based multi-GPU high-order code for hypersonic aerothermodynamics. Computer Physics Communications 255 (2020), 107262.
[145] Mario Di Renzo. 2022. HTR-1.3 solver: Predicting electrified combustion using the hypersonic task-based research solver. Computer Physics Communications 272 (2022), 108247.
[146] Seema Mirchandaney, Alex Aiken, and Elliott Slaughter. 2024. Speaking Pygion: Experiences Writing an Exascale Single Particle Imaging Code. In Workshop on Asynchronous Many-Task Systems and Applications. Springer, 1–8.
[147] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond data and model parallelism for deep neural networks. Proceedings of Machine Learning and Systems 1 (2019), 1–13.
[148] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 559–578.
[149] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. 2022. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 267–284.
[150] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. DISTAL: the distributed tensor algebra compiler. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 286–300.
[151] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. Proceedings of the ACM on Programming Languages 1, OOPSLA (2017), 1–29.
[152] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: compiling distributed sparse tensor computations. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 1–15.
[153] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: a high-productivity programming language for HPC with logical regions. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1–12.
[154] E. Slaughter. 2017. Regent: A High-Productivity Programming Language for Implicit Parallelism with Logical Regions. Ph.D. Dissertation. Stanford University.
[155] Elliott Slaughter and Alex Aiken. 2019. Pygion: Flexible, scalable task-based parallelism with python. In 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM). IEEE, 58–72.
[156] Michael Bauer and Michael Garland. 2019. Legate NumPy: accelerated and distributed array computing. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 23.
[157] A. Bouteiller, T. Herault, Q. Cao, J. Schuchart, and G. Bosilca. 2025. PaRSEC: Scalability, flexibility, and hybrid architecture support for task-based applications in ECP. IJHPCA 39, 1 (Jan. 2025), 147166. DOI: http://dx.doi.org/10.1177/10943420241290520
[158] Q. Cao, G. Bosilca, W. Wu, D. Zhong, A. Bouteiller, and J. Dongarra. 2020. Flexible Data Redistribution in a Task-Based Runtime System. In 2020 IEEE CLUSTER. 221–225. DOI: http://dx.doi.org/10.1109/CLUSTER49012.2020.00032
[159] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. 2015. Hierarchical DAG Scheduling for Hybrid Distributed Systems. In 2015 IEEE International Parallel and Distributed Processing Symposium. 156–165.
[160] George Bosilca, Aurélien Bouteiller, Thomas Hérault, Pierre Lemarinier, Narapat Ohm Saengpatsa, Stanimire Tomov, and Jack J. Dongarra. 2011. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. In 2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011. IEEE Computer Society, 395–402. DOI: http://dx.doi.org/10.1109/CLUSTER.2011.51
[161] Thomas Hérault, Yves Robert, George Bosilca, and Jack J. Dongarra. 2019. Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC. In 10th IEEE/ACM Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2019, Denver, CO, USA, November 18, 2019. IEEE, 33–41. DOI: http://dx.doi.org/10.1109/SCALA49573.2019.00010
[162] Aurelien Bouteiller, Qinglei Cao, Joseph Schuchart, and Thomas Hérault. 2025. Comparing and Contrasting User and Runtime Directed Data Placement Strategies for Owner-Compute, Multi-accelerator Distributed Task Based Scheduling. In Asynchronous Many-Task Systems and Applications - Third International Workshop, WAMTA 2025, St. Louis, MO, USA, February 19-21, 2025, Proceedings (Lecture Notes in Computer Science), Patrick Diehl, Qinglei Cao, Thomas Hérault, and George Bosilca (Eds.), Vol. 15690. Springer, 140–153. DOI: http://dx.doi.org/10.1007/978-3-031-97196-9_12
[163] Chong Peng, Justus A Calvin, Fabijan Pavosevic, Jinmei Zhang, and Edward F Valeev. 2016. Massively parallel implementation of explicitly correlated coupled-cluster singles and doubles using TiledArray framework. The Journal of Physical Chemistry A 120, 51 (2016), 10231–10244.

[164] Chong Peng, Cannada A Lewis, Xiao Wang, Marjory C Clement, Karl Pierce, Varun Rishi, Fabijan Pavošević, Samuel Slattery, Jinmei Zhang, Nakul Teke, et al. 2020. Massively Parallel Quantum Chemistry: A high-performance research platform for electronic structure. The Journal of Chemical Physics 153, 4 (2020).

[165] Thomas Herault, Yves Robert, George Bosilca, and Jack J. Dongarra. 2019. Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC. In ScalA@SC. IEEE, 33–41. DOI:http://dx.doi.org/10.1109/SCALA49573.2019.00010

[166] Thomas Herault, Yves Robert, George Bosilca, Robert J. Harrison, Cannada A. Lewis, Edward F. Valeev, and Jack J. Dongarra. 2021. Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure. In International Parallel and Distributed Processing Symposium. IEEE, 537–546. DOI:http://dx.doi.org/10.1109/IPDPS49936.2021.00062

[167] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G. Genton, David E. Keyes, Hatem Ltaief, and Ying Sun. 2022. Reshaping Geostatistical Modeling and Prediction for Extreme-Scale Environmental Applications. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis. 1–12. DOI:http://dx.doi.org/10.1109/SC41404.2022.00007

[168] Sameh Abdulah, Allison H. Baker, George Bosilca, Qinglei Cao, Stefano Castruccio, Marc G. Genton, David E. Keyes, Zubair Khalid, Hatem Ltaief, Yan Song, Georgiy L. Stenchikov, and Ying Sun. 2024. Boosting Earth System Model Outputs And Saving PetaBytes in Their Storage Using Exascale Climate Emulators. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '24). IEEE Press, Article 2, 12 pages. DOI:http://dx.doi.org/10.1109/SC41406.2024.00008

[169] Hatem Ltaief, Rabab Alomairy, Qinglei Cao, Jie Ren, Lotfi Slim, Thorsten Kurth, Benedikt Dorschner, Salim Bougouffa, Rached Abdelkhalak, and David E. Keyes. 2024. Toward Capturing Genetic Epistasis From Multivariate Genome-Wide Association Studies Using Mixed-Precision Kernel Ridge Regression. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '24). IEEE Press, Article 6, 12 pages. DOI:http://dx.doi.org/10.1109/SC41406.2024.00012

[170] Robert J. Harrison, Gregory Beylkin, Florian A. Bischoff, Justus A. Calvin, George I. Fann, Jacob Fosso-Tande, Diego Galindo, Jeff R. Hammond, Rebecca Hartman-Baker, Judith C. Hill, Jun Jia, Jakob S. Kottmann, M-J. Yvonne Ou, Junchen Pei, Laura E. Ratcliff, Matthew G. Reuter, Adam C. Richie-Halford, Nichols A. Romero, Hideo Sekino, William A. Shelton, Bryan E. Sundahl, W. Scott Thornton, Edward F. Valeev, Álvaro Vázquez-Mayagoitia, Nicholas Vence, Takeshi Yanai, and Yukina Yokoi. 2016. MADNESS: A Multiresolution, Adaptive Numerical Environment for Scientific Simulation. SIAM Journal on Scientific Computing 38, 5 (2016), S123–S142. DOI:http://dx.doi.org/10.1137/15M1026171

[171] G I Fann, R J Harrison, G Beylkin, J Jia, R Hartman-Baker, W A Shelton, and S Sugiki. 2007. MADNESS applied to density functional theory in chemistry and nuclear physics. Journal of Physics: Conference Series 78, 1 (jul 2007), 012018. DOI:http://dx.doi.org/10.1088/1742-6596/78/1/012018

[172] solverstack 2025. SolverStack @ Inria Bordeaux. (2025). https://https://solverstack.gitlabpages.inria.fr/.

[173] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. 2010. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In GPU Computing Gems, Wen mei W. Hwu (Ed.). Vol. 2. Morgan Kaufmann. https://inria.hal.science/inria-00547847

[174] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. 2011. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In 25th IEEE International Parallel & Distributed Processing Symposium. Anchorage, United States. DOI:http://dx.doi.org/10.1109/IPDPS.2011.90

[175] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. 2011. LU Factorization for Accelerator-based Systems. In 9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11). Sharm El-Sheikh, Egypt. https://inria.hal.science/hal-00654193

[176] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. 2017. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. IEEE Transactions on Parallel and Distributed Systems (2017). DOI:http://dx.doi.org/10.1109/TPDS.2017.2766064

[177] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. Parallel Computing 28, 2 (2002), 301–321. https://inria.hal.science/inria-00346017

[178] Pascal Hénon, Pierre Ramet, and Jean Roman. 2008. On finding approximate supernodes for an efficient ILU(k) factorization. Parallel Computing 34 (2008), 345–362. https://inria.hal.science/inria-00346018

[179] Grégoire Pichon, Mathieu Faverge, Pierre Ramet, and Jean Roman. 2017. Reordering Strategy for Blocking Optimization in Sparse Linear Solvers. SIAM Journal on Matrix Analysis and Applications 38, 1 (2017), 226 – 248. DOI:http://dx.doi.org/10.1137/16M1062454

[180] Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, and Jean Roman. 2018. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. International Journal of Computational Science and Engineering 27 (July 2018), 255 – 270. DOI:http://dx.doi.org/10.1016/J.JOCS.2018.06.007

[181] Alfredo Buttari. 2012. Fine Granularity Sparse QR Factorization for Multicore Based Systems. In Applied Parallel and Scientific Computing, Kristján Jónasson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 226–236.

[182] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. 2016. Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems. ACM Trans. Math. Softw. 43, 2, Article 13 (Aug. 2016), 22 pages. DOI:http://dx.doi.org/10.1145/2898348

[183] Stojce Nakov, Emmanuel Agullo, Alfredo Buttari, Luc Giraud, and Gilles Marait. 2025. Scalable Domain Decomposition Methods for Large Sparse Linear Systems on Modern Architectures. In Sparse Days 2025. Toulouse, France. https://inria.hal.science/hal-05248757

[184] Emmanuel Agullo, Luc Giraud, and Louis Poirel. 2019. Robust preconditioners via generalized eigenproblems for hybrid sparse linear solvers. SIAM Journal on Matrix Analysis and Applications 40, 2 (2019), 417–439. DOI:http://dx.doi.org/10.1137/17M1153765

[185] Pierre Blanchard, Olivier Coulaud, Eric Darve, and Alain Franc. 2016. FMR: Fast randomized algorithms for covariance matrix computations. Platform for Advanced Scientific Computing (PASC). (June 2016). https://hal.science/hal-01334747 Poster.

[186] Emmanuel Agullo, Olivier Coulaud, Alexandre Denis, Mathieu Faverge, Alain Franc, Jean-Marc Frigerio, Nathalie Furmento, Adrien Guilbaud, Emmanuel Jeannot, Romain Peressoni, Florent Pruvost, and Samuel Thibault. 2022. Task-based randomized singular value decomposition and multidimensional scaling. Research Report RR-9482. Inria Bordeaux - Sud Ouest ; Inrae - BioGeCo. 37 pages. https://inria.hal.science/hal-03773985

[187] Olivier Coulaud, Alain Franc, Jean-Marc Frigerio, Lucas Leandro, Romain Peressoni, and Florent Pruvost. 2025. cppdiodon. https://gitlab.inria.fr/diodon/cppdiodon. (Feb. 2025). https://hal.science/hal-04971369 Version 0.4. CeCILL-C Free Software License Agreement. Keywords: Randomized Linear Algebra; StarPU; MPI; HPC; Principal Component Analyses (PCA); Multidimensional Scaling (MDS); Multivariate Analysis.

[188] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre Brenner. 2017. Design and Analysis of a Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping. International Journal of Computational Science and Engineering (2017), 1 – 22. DOI:http://dx.doi.org/10.1016/j.jocs.2017.03.008

[189] Alice Lasserre et al. 2024. Multi-Criteria Mesh Partitioning for an Explicit Temporal Adaptive Task-Distributed Finite-Volume Solver.
In The 25th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2024). San Francisco, United States, 10. https://inria.hal.science/hal-04403209 Best Paper Award.

[190] Bérenger Bramas. 2020. TBFMM: A C++ generic and parallel fast multipole method library. Journal of Open Source Software 5, 56 (Dec. 2020), 2444. DOI:http://dx.doi.org/10.21105/joss.02444

[191] Aleksandr Mikhalev, Aleksandr Katrutsa, Konstantin Sozykin, and Ivan Oseledets. 2025. NNTile: a machine learning framework capable of training extremely large GPT language models on a single node. (2025). arXiv:cs.LG/2504.13236 https://arxiv.org/abs/2504.13236

[192] J. Spinti, J. Thornock, E. Eddings, P. Smith, and A. Sarofim. 2008. Heat transfer to objects in pool fires. In Transport Phenomena in Fires. WIT Press, Southampton, U.K.

[193] M. Berzins et al. 2016. Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices. SIAM Journal on Scientific Computing 38, 5 (2016), S101–S122. http://www.sci.utah.edu/publications/Ber2015a/detonationsiam16-2.pdf

[194] S. Kumar et al. 2018. Scalable Data Management of the Uintah Simulation Framework for Next-Generation Engineering Problems with Radiation. In Supercomputing Frontiers. Springer International Publishing, 219–240. http://www.sci.utah.edu/publications/Kum2018a/Scalable_Data_Management_of_the_Uintah_Simulation_.pdf

[195] Jennifer P. Spinti et al. 2022. Atikokan Digital Twin: Machine learning in a biomass energy system. Applied Energy 310 (2022), 118436. DOI:http://dx.doi.org/10.1016/j.apenergy.2021.118436

[196] Nikhil Jain et al. 2016. OpenAtom: Scalable Ab-Initio Molecular Dynamics with Diverse Capabilities. In International Supercomputing Conference (ISC HPC '16 (to appear)).

[197] Wojciech T. Soowski, Martin Berzins, William M. Coombs, James E. Guilkey, Matthias Möller, Quoc Anh Tran, Tito Adibaskoro, Seyedmohammadjavad Seyedan, Roel Tielen, and Kenichi Soga. 2021. Chapter Two - Material point method: Overview and challenges ahead. In Advances in Applied Mechanics, Stéphane P.A. Bordas and Daniel S. Balint (Eds.). Vol. 54. Elsevier, 113–204. DOI:http://dx.doi.org/10.1016/bs.aams.2020.12.002

[198] Bita Hasheminezhad, Shahrzad Shirzad, Nanmiao Wu, Patrick Diehl, Hannes Schulz, and Hartmut Kaiser. 2020. Towards a scalable and distributed infrastructure for deep learning applications. In 2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS). IEEE, 20–30.

Received 8 December 2025; revised 28 July 2026; accepted 7 August 2026