LA-UR-21-20928
Approved for public release; distribution is unlimited.

Title: TBAA20: Task-Based Algorithms and Applications

Author(s): Demeshko, Irina P.
Diehl, Patrick
Adelstein-Lelbach, Bryce
Buch, Ronak
Kaiser, Hartmut
Kale, Laxmikant (Sanjay)
Khatami, Zahra
Koniges, Alice
Shirzad, Shahrzad

Intended for: Report

Issued: 2021-02-02

Los Alamos
NATIONAL LABORATORY
EST. 1943

TBAA20: Task-Based Algorithms and Applications

Bryce Adelstein-Lelbach
NVIDIA
Santa Clara, CA, USA
brycelelbach@gmail.com

Ronak Buch
Department of Computer Science
University of Illinois at Urbana-Champaign
Champaign, IL, USA
rabuch2@illinois.edu

Irina P. Demeshko
Los Alamos National Laboratory
New Mexico, USA
irina@lanl.gov

Patrick Diehl
Center of Computation & Technology
Louisiana State University
Baton Rouge, LA, USA
pdiehl@cct.lsu.edu

Hartmut Kaiser
Department of Electrical Engineering
Louisiana State University
Baton Rouge, Louisiana
hkaiser@cct.lsu.edu

Laxmikant (Sanjay) Kale
Department of Computer Science
University of Illinois at Urbana-Champaign
Champaign, IL, USA
kale@illinois.edu

Zahra Khatami
Oracle
Redwood City, CA, USA
zahra.k.khatami@oracle.com

Alice Koniges
Maui High Performance Computing Center
University of Hawaii
Honolulu, HI, USA
aekoniges@lbl.gov

Shahrzad Shirzad
Center of Computation & Technology
Louisiana State University
Baton Rouge, Louisiana
sshirz1@lsu.edu

Abstract

The new challenges posed by Exascale system architectures have resulted in difficulty achieving a desired scalability using traditional distributed-memory runtimes. Task-based programming models show promise in addressing these challenges, providing application developers with a productive and performant approach to programming on next generation systems. Empirical studies show that task-based models can overcome load-balancing issues that are inherent to traditional distributed-memory runtimes, and that task-based runtimes perform comparably to those systems when balanced. This panel is designed to explore the advantages of task-based programming models on modern and future HPC systems from an industry, university, and national lab perspective. It aims at gathering application experts and proponents of these models to present concrete and practical examples of using task-based runtimes to overcome the challenges posed by Exascale system architectures. This report describes the objectives, activities, and outcomes of the panel TBAA: Task-Based Algorithms and Applications which was held at the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 20) on November 18, 2020.

Keywords: task-based algorithms · asynchronous many-task runtime systems · Charm++ · Julia · HPX · OpenMP

1 Introduction

High Performance Computing Systems (HPC) are moving heavily toward many-core processors with deep memory hierarchies. Achieving maximum parallelism level in model supercomputers requires maintaining load balance among all available resources to avoid any potential resource starvation while managing overheads and latencies. With memory being distributed, this challenge becomes even more complex. On the other hand, the most commonly used distributed-memory programming models are too explicit, making users responsible for overlapping communication and computation, resulting in difficulty achieving a desired scalability on these modern supercomputers.

Emerging task-based parallel programming models shield the programmer from the challenging task of identifying and managing parallelism by delegating this task to the runtime system. Asynchronous many-task systems (AMT) are based on a new paradigm addressing Exascale challenges by providing a productive and performant approach to programming on next generation systems. Empirical studies show that AMT-based models can overcome load-balancing issues that are inherent to traditional distributed-memory runtimes, and that AMT-based runtimes can perform comparably to those systems when balanced. In addition to maintaining load-balancing, AMT-based models can achieve a desired parallelism performance by directly supporting asynchronous execution, reducing the number of threads which need to synchronize, and hiding communication latencies behind useful work, thereby increasing system utilization.

Even though dozens of different task-based systems exist today and are actively used for parallel and high-performance computing, they are not commonly used in HPC applications and algorithm implementations. However, for certain types of irregular, task-based applications, it has been shown that task-based programming models are beneficial. This panel is designed to explore the advantages of task-based programming models over other approaches.

In addition, it is designed to draw more attention to new programming models by demonstrating results from utilizing task-based programming models in scientific simulations and algorithms.

The panel was moderated by Patrick Diehl (Louisiana State University) and the following panelists contributed to the discussion: Laxmikant V. Kale (University of Illinois), Irina P. Demeshko (Los Alamos National Laboratory), Bryce Adelstein Lelbach (NVIDIA), Hartmut Kaiser (Louisiana State University), Zahra Khatami (Oracle), Keno Fischer (Julia Computing Inc), and Alice Koniges (University of Hawaii). The panels was kicked-off by Patrick Diehl with the introduction of the panelists. In the first talk Bryce Adelstein-Lelbach introduces the concept of asynchronous many-task runtime systems, see Section 1.1. Next a brief overviews of four AMTs were given: Charm++ by Laxmikant V. Kale, see Section 1.2; HPX by Hartmut Kaiser, see Section 1.3; and OpenMP by Alice Koniges, see Section 1.4. For a comparative review on AMTs, we refer to [1]. In the remaining panel time, the panel's chosen questions in Section 2 were discussed. Section 3 concludes the panel and phrases some of the audience's questions.

1.1 Asynchronous many-task system

In the Asynchronous Many Task Model, the task is the central abstraction. A task can be thought of as a function call - a function to be invoked, a set of arguments to invoke it with, and possibly metadata. Some AMT frameworks use a more object oriented model, where a task may invoke a member function on a specific object.

Tasks may have dependencies, either dependencies on other tasks or dependencies on data expressed through the arguments that they take. These dependencies are either expressed explicitly by the programmer or implicitly inferred based on usage. Task dependencies form an execution graph that is used to make scheduling decisions. This graph may be fully constructed prior to execution or dynamically discovered as execution proceeds.

Tasks are typically oversubscribed relative to physical hardware execution resources - the number of tasks is usually much greater than the number of physical hardware cores that are executing said tasks. Tasks can have drastically varying execution times, from a few cycles to minutes or hours. AMT frameworks are typically designed to efficiently handle these varying execution times, making them an excellent fit for computational problems that have a non-uniform or dynamically evolving workload.

In AMT frameworks, a runtime system is responsible for deciding when and where to execute tasks based on the task dependency graph. These runtime systems often support a variety of scheduling strategies, hardware, and underlying execution frameworks.

Most AMT frameworks also support distributed execution - there is no distinction between executing work on a core local to the current system and executing work on a core on another node. Such AMTs usually have a global object space that makes some or all data accessible by all execution resources.

AMT offers a unified programming model for heterogeneous-, thread- and node-level parallelism. The task abstraction is a natural fit for GPU and distributed programming, where the destination execution resource may not be able to access the memory of the sender. The global object space and the task execution graph allow an AMT framework to determine which data needs to made accessible to the execution resource

where a task will be executed.

1.2 Charm++

Charm++ [2, 3, 4] is a parallel programming system based on the notion of migratable objects. A computation specified by a Charm++ program consists of a large number of C++ objects that interact with each other via asynchronous method invocations. The objects are organized into multiple collections. Objects within each collection are individually identified by their index. The index may be a multidimensional space, which may be densely or sparsely populated.

When an object wants to communicate with another object, say `A[23]`, it does so by simply making a method invocation `A[23].foo(…)`. This specifies the object at index 23 of the collection named `A` as the target of the method invocation. The method invocation is asynchronous, meaning that the call returns immediately while invocation is sent towards that object. The programmer does not need to know where that particular object is located; it is the responsibility of the Charm++ runtime system to serialize the method invocation into a message, locate the processor that currently houses the target object using its distributed location management system, and send the message to that processor, where it is enqueued into a pool of messages served by a user-space scheduler. When the scheduler eventually dequeues the message, it confirms that the target object is still located in the same processor, and invokes the specified method `foo(...)` with the serialized arguments from the message. The invocation is allowed to run to completion, at which point more messages may be generated for other objects, and the scheduler is free to select the next method invocation from its pool. This asynchronous model also means global operations such as reductions are non-blocking, delivering the results when they are ready while allowing other computations to proceed. This avoids unnecessary synchronization and resultant performance loss due to transient imbalances.

An introspective and adaptive runtime system is responsible for deciding which objects live on which processor; it may change this mapping of objects to processors during program execution. The programmer does not need to be aware of this "migration" of objects because their communication is specified with a logical object as the target, not a processor. Since the runtime system is in charge of scheduling method executions and managing locations, it is aware (via instrumentation inserted by the runtime system) of how much execution time, i.e. computational load, each object is using. Since it is in charge of mediating communication between objects, it also knows which objects talk to other objects and with what communication intensity. With continuous monitoring of the evolution of load imbalances, the system decides when to balance the load, and, using one of several load balancing strategies, computes a new mapping of objects to processors (usually minimizing object migration by keeping objects on the same processor as much as possible), and migrates the objects to improve load balance and reduce communication costs [5].

This adaptive capability makes several online optimizations possible. Processor loads may evolve either due to application-related factors or machine-related factors. In either case, the system is able to improve performance. The same serialization mechanism used for migrating objects is used by the runtime to automatically checkpoint the state of a program to the file system [6]. Further, an in-memory checkpoint / restart system combined with automatic fault detection allows the system to continue execution even when individual nodes fail, distributing objects from the failed node to the remaining nodes within the original job

allocation. Runtime elasticity is supported, where the number of nodes allocated to a job can be changed during program execution. Many energy-related optimizations are also enabled by the adaptive runtime system [7].

There are several related projects built on top of Charm++, including charm4py [8], a Python interface to Charm++ and Adaptive MPI [9], and an MPI implementation where the notion of a rank is migratable and virtualized. There are more ranks than processors, allowing it to easily utilize Charm++ runtime services, like load balancing and fault tolerance.

1.3 HPX - The C++ Standard Library for Parallelism and Concurrency

Although current programming models like OpenMP and MPI make it possible for programmers to gain better parallelism through more sophisticated means, such as overlapping computation with communication and using active messages, they impose insufficient parallelism by design. The enforced barrier at the end of the parallel loops in OpenMP, and the global communication barrier after each time step for MPI in distributed applications are some of the examples of this issue[10]. The other drawback of these models is their tendency for over-synchronization. One example could be the lock-step between nodes or ranks which causes them to be on the same time step. This makes it difficult to decouple the time stepping on different nodes in scientific simulations. The other issue is insufficient coordination between on-node and off-node parallelization techniques available to us.

HPX [11] is designed to expose a coherent and uniform API to users that does not require additional effort to overcome the mentioned problems. Using the API exposed by HPX and the programming model that comes with it would intrinsically expose features such as overlapping communication with computation, using active messages, hiding latency, and load balancing on and across the nodes.

HPX creates lightweight user-level threads with fast context switching while its thread manager ensures high utilization of resources and scalability through work stealing, work sharing, and customized scheduling policies. Active Global Address Space (AGAS) implemented within HPX enables a uniform addressing of local and remote objects by assigning Global IDs (GID) to objects, facilitating object migration for the purpose of load balancing[11]. HPX also provides an integrated framework of globally accessible performance counters for performance measurement, and analysis, with the ability to impose runtime adaptivity, APEX [12].

HPX is conformant to the C++ standard library while extending it to support distributed cases [13]. This makes it very easy for developers with C++ knowledge to utilize HPX.

1.4 OpenMP

OpenMP gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on a wide range of platforms. In 2008, OpenMP Version 3.0 added the task construct. In 2013, Version 4.0 supports taskgroup and task dependencies. In 2015, Version 4.5 supports taskloops and task priorities. In 2018, Version 5.0 supports task reductions, task affinity and detached tasks. Additionally,

OpenMP tasking supports data-locality-aware scheduling, futures, groups, and cancellation. OpenMP tasking directives allow asynchronous programming to improve load-balancing computation, to help orchestrate work between multicores and accelerators, and to support multi-level parallelism. In OpenMP, a task is executed immediately or deferred for later execution. Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution. In OpenMP, tasks can be nested: i.e. a task may itself generate tasks. To learn how the new versions of the OpenMP specification are produced and approved see[1]. To get an introduction to OpenMP tasking and other key features of OpenMP see [14]. For more details on OpenMP through Version 4.5 see [15].

2 Panel's Chosen Questions

2.1 For what class of applications is an AMT paradigm the best solution for achieving scalable execution?

Irina AMT models will suit the best to the applications that have independent pieces of work (computations) that can be performed asynchronously.

Zahra From my experience, AMT becomes pretty useful, especially for the applications having an irregular dependency pattern. AMTs are flexible and they are able to capture these different data dependencies.

Sanjay AMT systems shine when flexible dynamic scheduling of work is necessary to obtain good performance and to facilitate modular expression of control. Transfer of control from one piece of work to another (maybe one module to another, for example) is typically explicitly specified by the programmer in traditional parallel programming systems. AMTs typically do it differently. In Charm++, control transfers are dictated by the availability of data, and this is not just during one portion of the program, but even across modules.
Charm++ also deals with across-node performance issues via object migration. Overall, AMTs tend to be beneficial for irregular and/or dynamic scenarios, arising out of application behavior or hardware variability. The latter is worthwhile to note: even for regular applications with no dynamic variations, the hardware may create variability of its own, due to reasons such as chip-to-chip process variation, thermal behavior especially with DVFS and automatic underclocking and overclocking leading to frequency variability, differential use and load and availability of caches, etc.

2.2 How does the C++ language standard impact efficiency and complexity of programming AMT systems?

Alice Standards are critical for advancing all programming models. It is important that standard committees consist of a mix of application developers, compiler writers, and architecture savvy vendors.

[1] `https://www.openmp.org/resources/refguides/`

Sanjay The team working on maintaining and developing Charm++ is very enthusiastic about C++ and using many of its modern features. I have a couple of reservations about C++: first, there are times when low-level C++ abstractions tend to hide efficiency issues. Second, I am concerned about the continual increase in the complexity of the C++ language standard.
Charm++ also has a unique problem with being on the cutting edge of C++, arising from our success: backward compatibility. As a successful system with production users on all supercomputers, we could not even afford to use C++11 in Charm++ because one of the major supercomputer sites was not able to upgrade to the latest compilers. Going with generic compilers, rather than vendor-supplied compilers, was not quite an option because of the performance loss. However, with the decommissioning of that particular machine, we have unshackled ourselves from this issue and are now using many C++11 features. Moving to C++11 has been very helpful, particularly for portability; being able to use things like `std::atomic`, `std::mutex`, etc. have allowed the runtime code to move away from a maze of verbose, error-prone, platform-specific `#ifdefs`.
One area where the standard is lacking is in reflection. Guaranteed unique identifiers for types in `std::type_info` between invocations (important in the usual SPMD context) and ways to programmatically access the type information of class contents would reduce the burden we put on the user for serialization and allow stronger typing to be used throughout the system.
Overall, modern C++ helps improve the level of abstraction. Charm++'s inherent need for modern features is light, and so we can develop it in such a way that users can use modern features as needed, but the runtime does not force them to.

Hartmut Due to the compliance of HPX with the latest C++ standards, it integrates well with other C++ standards conforming libraries and facilities. From my experience, HPX has a smooth learning curve for users with prior C++ experiences since they are aware of many concepts from the C++ Standard Library. Another factor is the portability of code and performance, e.g. `std::future` and `std::async` can easily be replaced by `hpx::future` and `hpx::async` without changing other parts of the code base. Moreover, HPX extends many of the standard facilities related to parallelism and concurrency to the distributed case, thus allowing to write code that is syntactically and semantically equivalent for local and remote operations.

2.3 What are the differences between HPX, Julia, and Charm++ AMT paradigms, and how can these differences affect the parallel performance?

Irina From the application developer's perspective, Charm++ and HPX programming models are similar. HPX is developed as an abstraction to C++, Charm++ is more as a standalone library.

Hartmut In my opinion, Charm++ and HPX share common principles and concepts. However, HPX is in compliance with the C++ standard and feels more like an extension of the C++ Standard Library. The unique characteristic of HPX is this compliance with respect to other distributed AMTs.

Sanjay One of the features that distinguishes Charm++ is its emphasis on runtime adaptation based on continuous introspection. The idea of runtime adaptation itself is not new; researchers have long talked about sensors and actuators in parallel computing. However, traditional programming models do not create adequate levers for significant adaptation that will impact performance. Overdecomposition creates such levers, by virtue of having the supply of a large number of objects that can be moved around across the system, and having control over which, of many, ready tasks (typically, asynchronous method invocations, in addition to explicitly created tasks) to schedule next. As a result, the runtime system can interject itself in the execution to instrument and monitor object execution times and inter-object communication patterns, as well as processor and node loads and bandwidth utilization. Using multiple strategies, including some based on machine learning, it can actuate some of its levers (by migrating objects, and prioritizing tasks scheduling, for example) to significantly impact performance.
Charm++ also leverages what we call "the principle of persistence" which is a heuristic that characterizes the behavior of parallel programs. Once a program is expressed in terms of logical objects, the computational load and communication patterns among these objects tend to persist from iteration to iteration, in spite of dynamic variation. This is only a heuristic, but it applies often enough to be useful. Movement of particles across cells may change the load, but only gradually. Adaptive mesh refinement may create abrupt changes in load, but typically only infrequently. In those situations, recent behavior can be used to predict the near future, allowing the runtime system to adapt to evolving behavior. Of course, when this prediction principle does not apply, the runtime can still use more dynamic methods. However, in an otherwise unfriendly environment of dynamically changing applications and hardware, this principle of persistence is often our ally.
Charm++ is different than some of the other systems, such as Julia or Legion/Regent, in another way: it does not rely on the compiler and is mostly a C++ library. The exceptions are: (a) interface files that are needed to generate serialization code and register classes and methods; and (b) a script-like notation called structured dagger which is used to express dependencies of tasks on data and incoming method invocations as well as program order.

2.4 Have these HPX, Julia, and Charm++ AMT paradigms performances been evaluated against scientific applications, AI, or any other frameworks? If so, how did those frameworks benefit using these AMT models compared to the traditional runtime systems?

Irina There were several publications that compare specific AMT programming model performance when implemented in a scientific application. However, there is no performance comparison between different AMT models. We believe that having a set of benchmarks implemented with different AMT backends is needed to fill the gap.

Zahra In one of the projects that I was involved during my PhD program at LSU, I implemented N-Body application with HPX, see [16]. N-Body problem is known to be communication intensive, significantly affecting the execution time, due to message latencies and overheads. Updating the information for each of the particles requires accessing the information about all other particles. So, full parallelization in both space and time has to be considered to achieve a maximum possible level of parallelism. Local and global synchronization barriers, communication overhead, and load balancing are the major challenges in achieving the maximum parallelism level using conventional parallel programming. The `future` concept as implemented in HPX is one possible way of enabling this full parallelization. In this project, HPX was used for both communication between different nodes and parallel processing within each node to develop a new parallel distributed N-Body application that meets these challenges. Thus, we were able to achieve better load balancing and lower overhead when compared to non-AMT paradigms.

Sanjay Several highly scalable CSE applications have been developed using Charm++. The most well-known is NAMD [17], for biomolecular simulations including the SARS-CoV-2 simulation, which won a Gordon Bell Award this year. Other examples include ChaNGa and SpECTRE for astronomy, and OpenAtom for quantum chemistry electronic structure calculations. A framework for structured adaptive refinement called Cello has been developed at UCSD using Charm++ along with the Enzo-P astrophysics application. Most of these have been scaled to the largest supercomputers and hundreds of thousands of cores, and performance has been on par or better than other applications in their class, especially when load balancing and asynchrony become important.

An early example of the benefit of asynchrony was provided by NAMD running on Pittsburg's Lemieux supercomputer [18]. OS-induced computational "noise" was just becoming a new challenge on parallel machines, especially for applications with fine-grain parallelism such as NAMD. As noise impacts different processors at different times, each timestep is delayed by a different processor in applications with global synchronization at each step. However, in NAMD, the noise felt in one step by one processor did not affect the overall performance because timestep boundaries are not global barriers.

A different answer to the question has to do with comparing AMT frameworks with each other. It is desirable to have a set of application problems (paper-and-pencil benchmarks, in the spirit of the original NAS benchmarks) with fully specified algorithms but no code, except perhaps a reference implementation. Different AMT programming systems can then implement those algorithms in the best way possible. This will allow comparison of performance as well as expressiveness. Of course, the suite of problems needs be selected to be representative and fair, which may be achieved by cooperatively collecting such a benchmark suite. HPC Challenge was an earlier attempt in this direction, but the benchmark selection there was more suited for comparing parallel machines. Intel's Parallel Research Kernels [19] constitute a more appropriate example, but one that is in need of expansion.

2.5 How does the supercomputing community benefit from using AMT on modern and future supercomputers?

Alice Effective use of AMT will require creation and implementation of important benchmarks and mini apps going forward.

Irina AMT programming models should help with data management and mapping computations to available resources on large heterogeneous HPC systems. This often results in better load balancing and overall performance. They also provide other benefits like integrated IO, fault-tolerance and interoperability with shared-memory libraries.

Sanjay Future hardware will present even larger static and dynamic performance variability, in part because of semiconductor process variation as we get to smaller feature sizes; AMTs can handle that well, without programmer intervention. However, even more significantly, as we get more powerful computers, scientists are likely to aim at strong scaling more and more. This leads to application-level adaptivity, via techniques such as dynamic adaptive mesh refinement, because ”refining everywhere” is no longer a reasonable option. This also plays to the advantage of AMT systems, especially when they are supported by adaptive runtime systems.

Hartmut Future hardware will provide more fine-grain parallelism by having more cores on a single device, e.g. GPUs or Arm64FX. However, the more hardware parallelism a device provides to the application, the better the application must scale to utilize all cores. To utilize as many cores as possible, an application must exhibit optimal strong and weak scaling behavior. The following fundamental factors influence scaling: Starvation occurs when there is insufficient concurrent work available to maintain high utilization of all resources. Latencies are imposed by the time-distance delay intrinsic to accessing remote resources and services. Overhead is work required for the management of parallel actions and resources on the critical execution path, which is not necessary in a sequential variant. Waiting for contention resolution is the delay due to the lack of availability of oversubscribed shared resources.
Asynchronous many-task systems address these limitations and enable the user to utilize the cores efficiently. Using HPX as an example, the following features address the SLOW limitations. HPX focuses on latency hiding instead of latency avoidance by intrinsically enabling overlap of computation and communication, by introducing finer-grain parallelism and constraint-based synchronization mechanisms. These things naturally help improve general resource utilization and thus increase parallel efficiency. Embracing fine-grained parallelism and using lightweight tasks instead of heavyweight (kernel) threads ensures the possibility of suspending a current task if some preconditions for its execution are not met (such as waiting for I/O or the result of a different task), seamlessly switching to any other task which can continue, and rescheduling the initial task after the required result has been calculated, making the implementation of latency hiding almost trivial. HPX also prefers constraint-based synchronization over global barriers. Any operation should proceed whenever the immediate preconditions for its execution are met, and only those. There is usually no need to wait for iterations of a loop to finish before the user can continue calculating other things; all that is needed is to complete the iterations that produce the required results for the next operation. The distributed nature of HPX and its API naturally enables preferring to move work to the data over moving data to work. The amount of bytes necessary to encode a certain operation is very often much smaller than the amount of bytes necessary for encoding the data the operation is performed upon. Thus, executing operations close to the data targeted by the operation enables adaptive locality control of data.

2.6 What hardware features are required in the next-generation processors to support AMT?

Zahra We know that hardware accelerators can help in speeding up CPUs. However, it is not easy to build a system with lots of accelerators, since this requires redesigning compilers and schedulers. Recognizing the granularity of the accelerators to support a different variety of the parallel algorithms is another challenge. We definitely need better task schedulers for the accelerators to be able to break down the applications into different tasks mapped on those accelerators.

Hartmut Here, the challenge is two-folded. First, the operations, e.g. copy data to / from the device, launching kernel, and compilation of the kernels, need to be seamlessly integrated within the asynchronous execution graph. All of these operations need to be wrapped into futures to synchronize them with the work on the CPU. In HPX we provide the `hpx::compute` [20] and hpx::cl [21] approaches for heterogeneous applications, see Octo-Tiger [22] as one example application. Second, with heterogeneous acceleration devices, like NVIDIA GPU, AMD GPU, AI devices, and special linear algebra devices, some layer of abstraction is necessary to avoid writing computational kernels for all of these devices and maintaining them for recent versions. Here, one example is Kokkos [23] to provide some abstractions which recently was extended to HPX as one backend programming model.

Sanjay Primarily, support for communication, scheduling, and event notifications will be useful to have in hardware. For example, hardware FIFOs, and possibly other priority queues, with the ability to asynchronously enqueue operations, with spillover to memory, and ability to probe multiple queues together will help. Support for futures, as mentioned by Hartmut, and other mechanisms for raising conditions, and triggering callbacks are also desirable. Additionally, being able to push pack / unpack, tag matching, and enqueueing operations to programmable NICs would improve performance and map well to AMTs since they are inherently asynchronous.

3 Conclusion

The panel gave an overview of the following AMTs: Charm++, HPX, Julia, and OpenMP, and for a broader comparative review, we refer to [1, 24, 25]. The panelists concluded the following for the panel's chosen questions: First, a suitable application for the AMT should involve flexible, dynamic scheduling of work, which means small independent tasks having irregular dependencies. Second, there were some reservations about the usage of recent modern C++ features due to backward compatibility. However, modern C++ features allow improving the level of abstraction. In addition, for HPX, the integration with other C++ standard confirming libraries is possible due its C++ standard compliance. Third, the Charm++ and HPX programming models share the same concepts. From the application developer's perspective, Charm++ feels more like a standalone library and HPX more like an extension of the C++ Standard Template Library. Fourth, AMT performance has been investigated in several scientific applications, however, there was no comparative performance analysis done between different AMT models. To compare the performance of AMTs, a set of benchmarks is needed. While there are many benchmarks available, these do not address the specific properties needed to stress the components of AMTs.

One suggestion from the audience was task-bench [26], which is one step in this direction. Currently, Julia and HPX are not yet integrated in the task-bench[2] suite. Other work in this area includes the Parallel Research Kernels [19]. Fifth, asynchronous many-task systems address certain limitations of currently widely used programming models as they help reduce global barriers during execution and synchronization overheads, in general; they naturally enable latency hiding and overlapping of computation with communication; they support introducing runtime-adaptive decision making that improves system utilization (adaptive scheduling, data placement, etc.); and last but not least, they allow data movement to be reduced by preferring to "send work to data" instead of "data to work". Sixth, the integration of next-generation processors within AMTs was discussed. One aspect here is the integration of these devices in the task scheduler and the asynchronous execution graph. Another aspect here is to provide some abstraction level to avoid having to implement each computational kernel in the different device-specific programming languages. To conclude, there was strong enthusiasm on the topic of task-based programming, with around 290 people attending the virtual panel, which ended in a fertile discussion session. A comparative benchmark suite of performance of the different AMTs seems to be of great interest, as well as the use of next-generation processor and acceleration cards in AMT systems.

References

[1] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. A Taxonomy of Task-based Parallel Programming Technologies for High-performance Computing. The Journal of Supercomputing, 74(4):1422–1434, 2018.

[2] Laxmikant Kale, Bilge Acun, Seonmyeong Bak, Aaron Becker, Milind Bhandarkar, Nitin Bhat, Abhinav Bhatele, Eric Bohm, Cyril Bordage, Robert Brunner, Ronak Buch, Sayantan Chakravorty, Kavitha Chandrasekar, Jaemin Choi, Michael Denardo, Jayant DeSouza, Matthias Diener, Harshit Dokania, Isaac Dooley, Wayne Fenton, Juan Galvez, Fillipo Gioachin, Abhishek Gupta, Gagan Gupta, Manish Gupta, Attila Gursoy, Vipul Harsh, Fang Hu, Chao Huang, Narain Jagathesan, Nikhil Jain, Pritish Jetley, Prateek Jindal, Raghavendra Kanakagiri, Greg Koenig, Sanjeev Krishnan, Sameer Kumar, David Kunzman, Michael Lang, Akhil Langer, Orion Lawlor, Chee Wai Lee, Jonathan Lifflander, Karthik Mahesh, Celso Mendes, Harshitha Menon, Chao Mei, Esteban Meneses, Eric Mikida, Phil Miller, Ryan Mokos, Venkatasubrahmanian Narayanan, Xiang Ni, Kevin Nomura, Sameer Paranjpye, Parthasarathy Ramachandran, Balkrishna Ramkumar, Evan Ramos, Michael Robson, Neelam Saboo, Vikram Saletore, Osman Sarood, Karthik Senthil, Nimish Shah, Wennie Shu, Amitabh B. Sinha, Yanhua Sun, Zehra Sura, Ehsan Totoni, Krishnan Varadarajan, Ramprasad Venkataraman, Jackie Wang, Lukasz Wesolowski, Sam White, Terry Wilmarth, Jeff Wright, Joshua Yelon, and Gengbin Zheng. The Charm++ Parallel Programming System, Aug 2019.

[3] Laxmikant V. Kale and Gengbin Zheng. Chapter 1: The Charm++ Programming Model. In Laxmikant V. Kale and Abhinav Bhatele, editors, Parallel Science and Engineering Applications: The Charm++ Approach, chapter 1, pages 1–16. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2013.

[4] 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. Parallel programming with migratable objects: Charm++ in practice. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, page 647–658. IEEE Press, 2014.

[5] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In Parallel and Distributed Computing for Symbolic and Irregular Applications, pages 167–181. World Scientific Publishing, 2000.

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

[6] E. Meneses, Xiang Ni, Gengbin Zheng, C. L. Mendes, and L. V. Kalé. Using migratable objects to enhance fault tolerance schemes in supercomputers. Parallel and Distributed Systems, IEEE Transactions on, 26(7):2061–2074, July 2015.

[7] Bilge Acun, Akhil Langer, Esteban Meneses, Harshitha Menon, Osman Sarood, Ehsan Totoni, and Laxmikant V Kalé. Power, reliability, and performance: One system to rule them all. Computer, 49(10):30–37, 2016.

[8] Juan J Galvez, Karthik Senthil, and Laxmikant Kalé. Charmpy: A python parallel programming model. In 2018 IEEE International Conference on Cluster Computing (CLUSTER), pages 423–433. IEEE, 2018.

[9] Laxmikant V. Kalé and Gengbin Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, Advanced Computational Infrastructures for Parallel and Distributed Applications, pages 265–282. Wiley-Interscience, 2009.

[10] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pages 1–11, 2014.

[11] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. HPX - The C++ Standard Library for Parallelism and Concurrency. Journal of Open Source Software, 5(53):2352, 2020.

[12] Kevin A Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D Malony, Thomas Sterling, and Rob Fowler. An autonomic performance environment for exascale. Supercomputing frontiers and innovations, 2(3):49–66, 2015.

[13] Standard ISO/IEC. ISO International Standard ISO/IEC 14882:2017(E) - Programming Language C++. Geneva, Switzerland: International Organization for Standardization (ISO), 2017.

[14] Timothy G Mattson, Yun He, and Alice Evelyn Koniges. The OpenMP Common Core: Making OpenMP Simple Again. MIT Press, 2019.

[15] Ruud Van der Pas, Eric Stotzer, and Christian Terboven. Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD. MIT Press, 2017.

[16] Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. A massively parallel distributed n-body application implemented with hpx. In 2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), pages 57–64. IEEE, 2016.

[17] James C. Phillips, David J. Hardy, Julio D. C. Maia, John E. Stone, João V. Ribeiro, Rafael C. Bernardi, Ronak Buch, Giacomo Fiorin, Jérôme Hénin, Wei Jiang, Ryan McGreevy, Marcelo C. R. Melo, Brian K. Radak, Robert D. Skeel, Abhishek Singharoy, Yi Wang, Benoît Roux, Aleksei Aksimentiev, Zaida Luthey-Schulten, Laxmikant V. Kalé, Klaus Schulten, Christophe Chipot, and Emad Tajkhorshid. Scalable molecular dynamics on cpu and gpu architectures with namd. The Journal of Chemical Physics, 153(4):044130, 2020.

[18] James C Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V Kalé. Namd: Biomolecular simulation on thousands of processors. In SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, pages 36–36. IEEE, 2002.

[19] Rob F Van der Wijngaart, Evangelos Georganas, Timothy G Mattson, and Andrew Wissink. A new parallel research kernel to expand research on dynamic load-balancing capabilities. In International Supercomputing Conference 2017, pages 256–274. Springer, 2017.

[20] Marcin Copik and Hartmut Kaiser. Using sycl as an implementation framework for hpx. compute. In Proceedings of the 5th International Workshop on OpenCL, pages 1–7, 2017.

[21] P. Diehl, M. Seshadri, T. Heller, and H. Kaiser. Integration of cuda processing within the c++ library for parallelism and concurrency (hpx). In 2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), pages 19–28, 2018.

[22] Gregor Daiß, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pfüger. 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, pages 1–37, 2019.

[23] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos. J. Parallel Distrib. Comput., 74(12):3202–3216, December 2014.

[24] Abhishek Kulkarni and Andrew Lumsdaine. A comparative study of asynchronous many-tasking runtimes: Cilk, charm++, parallex and am++. arXiv preprint arXiv:1904.00518, 2019.

[25] Reazul Hoque and Pavel Shamis. 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), pages 934–941. IEEE, 2018.

[26] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S Morris, Wonchan Lee, Qinglei Cao, et al. Task bench: A parameterized benchmark for evaluating parallel runtime performance. arXiv preprint arXiv:1908.05790, 2019.