A Survey of Programming Models and Frameworks for Heterogeneous Distributed Computing

Jacob Lambert
Computer Science Department, University of Oregon
CIS 630 Distributed Systems, Spring 2017
jlambert@cs.uoregon.edu

Abstract— **Programming computing clusters and supercomputers has traditionally assumed a homogeneous programming model, most commonly a hybrid MPI + OpenMP approach targeting only CPUs. Over the past decade, several different processor architectures have emerged as alternatives to traditional CPUs. Specialized hardware, for example GPUs, have several advantages over CPUs. A heterogeneous approach that can leverage the strengths of both CPUs and GPUs, as well as other types of processing units, can outperform a homogeneous approach. In this paper we survey several heterogeneous programming models and frameworks. We discuss the overarching themes in heterogeneous computing, describe each programming model, and summarize how these models fit into the overarching themes.**

Keywords— **Distributed Systems, Heterogeneous Computing, MPI, GASNet, OpenMP, CUDA, OpenCL, OpenMP, OpenACC**

I. Introduction

Historically, gains in computer processor performance have come from speeding up sequential executions and utilizing instruction-level parallelism, and the work has primarily focused on CPU performance. However, this trend has become unsustainable due to power demands and architecture designs. While most modern computing systems still heavily rely on CPUs for processing, over the last decade many alternative types of processors have become increasingly popular. The main alternate processing unit is the GPU, but other co-processors and accelerators include the Intel Xeon Phi, FPGAs, Cell processors.

These other types of specialized processors can have several performance advantages over traditional CPUs. For example, GPUs, which typically have many times more computing cores than a CPU, can realize greater overall throughput and better energy efficiency than a CPU. Similarly, FPGAs, which allow the hardware to be reconfigured to meet the needs of specific programs, outperform even GPUs in terms of throughput and energy efficiency for certain problems.

OpenMP [Other1] is undoubtedly the most popular solution for single-node shared memory multiprocessing using CPUs. However, there is limited OpenMP support for these alternative specialized processors. Because of the many advantages these accelerators provide, many programming frameworks have been developed to support these devices.

CUDA [Other2], one of the most popular APIs for general purpose GPU (GPGPU) programming, allows programming of Nvidia GPUs. The CUDA API is relatively low-level, and requires programmers to explicitly control thread execution on the GPU, and memory transfers to and from the device.

OpenCL [Other3] is an open-source alternative to CUDA, which allows programming of not only Nvidia GPUs, but also AMD GPUs, Intel FPGAs, Intel Xeon Phi devices, and many other types of accelerators. Like CUDA, OpenCL is a relatively low-level API requiring explicit programmer control.

OpenACC [Other4] is a more recent directive-based API, developed as a high-level alternative to CUDA and OpenCL. With OpenACC, the burden of controlling thread execution and device memory transfers is by default shifted to the compiler, although the programmer can still opt for more fine-grained control if desired.

These device APIs have seen wide adoption, and greatly enable and simplify programming of

heterogeneous systems. However, most of these frameworks are developed specifically for single-node systems, consisting of a single host CPU and a single device, or accelerator. Because large shared-memory systems are expensive and less scalable, to meet the demands of modern computing problems, most high performance systems are distributed in nature, consisting of tens or hundreds of nodes. For example, the Titan Supercomputer at ORNL has 18,688 nodes, each node consisting of a CPU and an Nvidia GPU.

For homogeneous computing across clusters and distributed systems, there is a well-established pattern of using a hybrid programming approach, involving the Message Passing Interface (MPI) [Other5] and OpenMP. MPI is responsible for handling the communication of data between distinct nodes. This hybrid approach, which utilizes MPI for communication and node-level parallelism and OpenMP for instruction-level parallelism, has been shown to produce successful, scalable solutions for homogeneous computing using only CPUs [Other6].

Because few accelerator devices support OpenMP, heterogeneous computing on distributed systems or clusters requires a different programming approach. Like the homogeneous case, it is possible to adopt a hybrid approach, replacing OpenMP with an appropriate device API. Many research projects have utilized MPI + CUDA or MPI + OpenCL approaches [Other7], [Other8], [Other9].

However, this hybrid approach to heterogeneous distributed computing introduces several inefficiencies. Creating programs using two separate programming models can be difficult, and programs created using MPI + CUDA or MPI + OpenCL can be harder to maintain, more complex, and less portable. Due to the manual management of all details in CUDA and OpenCL, it can become difficult and cumbersome to achieve asynchronous task parallelism, synchronize between kernels, and synchronize between hosts and GPUs or other accelerator devices. These increases in complexity can severely inhibit programmer productivity.

In addition to the programming complexity added, the hybrid approach can be subject to many performance inefficiencies as well. In the traditional MPI + X models, typically the host processor remains idle while an accelerator executes kernel codes. Similarly, accelerators are idle during MPI communication phases.

The programming models and frameworks surveyed in this paper attempt to remedy these negative aspects of MPI + X approaches in heterogeneous computing in one or more of the following ways:

- Provide higher-level communication abstractions over low-level MPI programming
- Provide higher-level device programming abstractions over CUDA and OpenCL
- Develop a single unified programming approach in contrast to the hybrid programming approach
- Overlap communication and kernel execution phases

The main goal of these improvements is to increase portability, maintainability, performance, and programmer productivity in distributed heterogeneous systems.

A. Overview

In section II, we briefly describe the primary components of eight notable programming approaches for heterogeneous computing on distributed systems. These approaches are listed in chronological order by release date. Many of the later approaches employ strategies first implemented in previous approaches.

Although each approach has the same end goal of distributed heterogeneous computing, each approach varies significantly in implementation strategy. Also, the publications on these works differ in which components receive the most focus in the written publications. These differences are reflected in the topics covered in their respective summary below. All figures in this section are originally from the respective publications.

In section III, we discuss the main overarching themes of the different programming approaches, and compare, contrast and categorize the different programming models and frameworks.

Finally, in section IV we conclude the survey and look at potential future directions of heteroge-

neous distributed programming.

II. Programming Models and Frameworks

A. StarPU

The first programming model we survey is StarPU [Main1], and more specifically StarPU-MPI [Main2], which is a multi-node extension to the original single-node implementation.

StarPU is an original runtime system with the goal of utilizing heterogeneous hardware without causing significant changes in programmer habits. StarPU aims to allow programmers to focus on high-level algorithm issues without being restrained by low-level scheduling issues.

StarPU presents a unified programming approach, and consists of a data-management facility and task execution engine. In the StarPU system, a central process maintains a queue of tasks, and a scheduler assigns these tasks to processing units, either a CPU or GPU.

One interesting aspect, covered in great detail in the StarPU publication, is StarPU's dynamic scheduling capabilities. Scheduling is a major component even in homogeneous systems. However, this problem is even more relevant in heterogeneous systems, where the execution time of a task can greatly differ depending on whether the task is assigned to a CPU or an accelerator.

StarPU allows different scheduling strategies to be employed at runtime, making the StarPU system a potential test bed for easily implementing and testing different schedulers for heterogeneous systems. The different strategies tested by the authors are listed in Fig 1. We briefly describe each strategy below.

Name	Policy description
greedy	Greedy policy with support for priorities
no-prio	Greedy policy without support for priorities
ws	Greedy policy based on Work Stealing
w-rand	Random weighted by processor speeds
heft-tm	Heterogeneous Earliest Finish Time [14]

Fig. 1
Scheduling policies implemented using StarPU interface

- **greedy**: In the default greedy approach, each time a CPU or GPU becomes available, it greedily selects the task with the highest programmer-defined priority from the task queue, and begins execution. This strategy can work well in some situations, but can experience poor performance in situations where certain tasks are well-suited to certain devices. With this greedy approach, these tasks may be completed by an inappropriate device, leading to lower performance.
- **no-prio**: Similar to the greedy approach, except without the programmer-defined priorities.
- **ws**: Also similar to the greedy approach, if processing unit has completed its task and no tasks are available in the queue, the processor attempts to steal work from other processes.
- **w-rand** In the random weighted scheduling approach, each processor is given a speed, or acceleration factor. The acceleration can be set by the programmer or measured using benchmarks. Then, tasks are scheduled according to these acceleration factors. For example, if a GPU has an acceleration factor four times higher than a CPU, the GPU should be scheduled approximately four times as much work. While this approach accounts for different processor speeds, it still suffers from the same problems as the greedy scheduler, in that tasks may not be assigned to the most appropriate processors.
- **heft-tm** The Heterogeneous Earliest Finish Time attempts solve the load-balancing issues presented by the other scheduling approaches. HEFT performs performance or cost modeling for each task across each available processing unit. Based on these performance models, the task can be assigned to the most appropriate processing unit, either a GPU or CPU. Ideally, tasks well-suited for CPUs will be assigned to CPUs, and the same for GPU-suited tasks. Figure 2 shows tasks with the relative speedups (CPU vs GPU) generated by the performance models, and how these tasks were scheduled to the CPU and GPU. We can see tasks better suited to the CPU are more commonly scheduled there, while GPU-suited tasks are most commonly scheduled to the GPU.

Kernel	Relative Speedup (1 GPU vs. 1 CPU)	Task balancing 3 CPUs	GPU
Pivot	×1.2	94.7 %	5.3 %
DGETRF	×6.7	0 %	100 %
DTRSM (lower)	×6.0	46.3 %	53.7 %
DTRSM (upper)	×6.2	45.3 %	54.7 %
DGEMM	×10.8	21.7 %	78.3 %

Fig. 2
SCHEDULING POLICIES IMPLEMENTED USING STARPU INTERFACE

While StarPU is designed for heterogeneous systems, it is limited to single-node applications. StarPU-MPI extends StarPU from single node to distributed heterogeneous systems.

StarPU-MPI accomplishes this by partitioning the task graph into multiple subgraphs. Data dependencies between nodes are handled by StarPU-MPI library calls, which call underlying MPI send and receives.

StarPU and StarPU-MPI offer a high-level uniform alternative to hybrid programming models for distributed heterogeneous computing, in that the programmer only needs to make StarPU API calls instead of both MPI + X calls.

B. *OmpSs*

Like StarPU, the implementation of OmpSs [Main3] has the goal of creating a unified programming approach for GPU-clusters. OmpSs extends the OpenMP programming model to include support for accelerator devices. Therefore, the OmpSs API is directive-based, resulting in a high-level alternative to CUDA and OpenCL. Simple pragmas can be used to perform data movement and synchronization. However, at the time of the publication, OmpSs still required kernels to be written using CUDA, as OmpSs only supports CUDA-enabled GPUs. An example of an OmpSs program including compiler directives is shown in Fig 3.

OmpSs is built on top of the Mercurium source-to-source compiler and the Nanos++ runtime library. The Mercurium compiler parses an input program, recognizes the OmpSs pragmas, and then transforms these pragmas into calls to the Nanos++

```
#pragma omp target device(cuda) copy_deps
#pragma omp task input([N] a) output([N] c)
void copy(double *a, double *c, int N);

#pragma omp target device (cuda) copy_deps
#pragma omp task input([N] c) output([N] b)
void scale(double *b, double *c, double scalar, int N);

#pragma omp target device (cuda) copy_deps
#pragma omp task input([N] a, [N] b) output([N] c)
void add(double *a, double *b, double *c, int N)

#pragma omp target device(cuda) copy_deps
#pragma omp task input([N] b, [N] c) output([N] a)
void triad(double *a, double *b, double *c,
                      double scalar, int N);

void stream (int N, double a[N], double b[N], int BSIZE)
{
  scalar = 3.0;
  for (k=0; k<NTIMES; k++)
  {
     int j;
     for (j=0; j<N; j+= BSIZE)
        copy (&a[j], &c[j], BSIZE);
     for (j=0; j<N; j+=BSIZE)
        scale (&b[j], &c[j], scalar, BSIZE);
     for (j=0; j<N; j+=BSIZE)
        add (&a[j], &b[j], &c[j], BSIZE);
     for (j=0; j<N; j+=BSIZE)
        triad (&a[j], &b[j], &c[j], scalar, BSIZE);
  }
}
```

Fig. 3
SCHEDULING POLICIES IMPLEMENTED USING STARPU INTERFACE

runtime library. It also generates device-dependent data to be used, including calls like cuda_malloc() or cuda_memcpy().

The Nanos++ runtime library is responsible for scheduling and executing parallel tasks. Like StarPU, Nanos++ internally represents tasks in a task dependency graph, which it passes to a scheduler. See Fig 4 for an overview of the OpmSs architecture.

The authors of the OmpSs publication also perform an extensive productivity evaluation, where they compare lines of code needed to program using a MPI + CUDA approach and the OmpSs approach. They show that the OmpSs approach to heterogeneous distributed computing significantly improves programmer productivity.

C. *Phalanx*

Phalanx [Main4] is another unified programming model for heterogeneous machines. However, unlike the previous models surveyed, Phalanx does not depend on MPI for the underlying communication. Phalanx instead relies on a Partitioned Global Address Space (PGAS) model, specifically GASNet [Other10].

In PGAS programming models, the individual

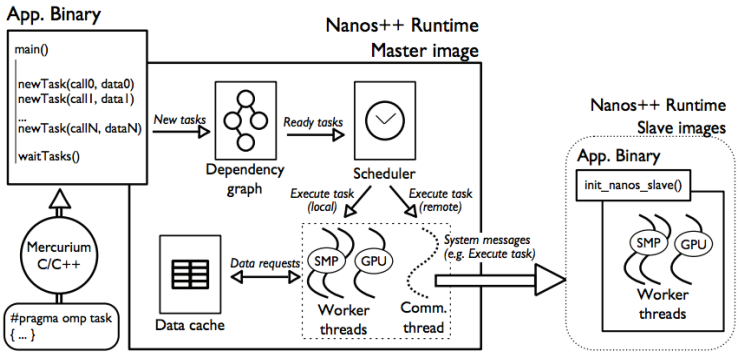

Fig. 4
OMPSS OVERVIEW FOR CLUSTERS OF GPUS

address spaces of distinct nodes and separate cores on each node are abstracted by a global address space. That is, each object during the distributed execution has a global address as well as a local address. This can allow programmers to write programs for distributed systems in a similar manner to shared memory systems. Popular PGAS languages include UPC, UPC++, Titanium, Chaple, and X10.

The Phalanx programming model is a C++ template library built on top of CUDA (GPU execution), OpenMP (CPU execution), and GASNet (communication). Because it is embedded in C++, Phalanx can easily be added to any existing C++ project. The goal of Phalanx is to make programming heterogeneous distributed systems simpler and more uniform. However, unlike some other models, Phalanx does not seek to provide high-level abstractions over communication and data movement.

Phalanx intentionally exposes the hardware memory hierarchy to allow expert programmers to precisely specify placements of tasks and data. Phalanx provides a **phalanx::place** object. This object, which represents the memory hierarchy, can be queried, filtered, and traversed to assign specific tasks to specific execution environments. However, less expert programmers can leave these assignment decisions up to the system.

D. SnuCL

SnuCL [Main5] is an extended OpenCL framework for heterogeneous CPU/GPU clusters. OpenCL can already be used to program both multicore CPUs and GPUs, however only on a single node. Additionally, the OpenCL system already implements the idea of a task or command queue, where tasks in the queue are assigned to either a CPU or device. The goal of SnuCL is to modify the OpenCL framework and extend the model for multi-node execution. The SnuCL abstracts a heterogeneous distributed system such that there is a single host node, along with one or more compute nodes.

The host node consists of a host thread and command scheduler. The host thread, which typically executes on a CPU, is responsible for running the host application. The host node is also responsible for queuing commands or tasks to the command queue, and managing the command scheduler. The command scheduler is responsible for assigning commands in the command queue to compute nodes.

Compute nodes consist of a command handler thread, and also device threads, one for each device in the compute node. The command handler thread processes commands from the host node, and reassigns the commands to a device thread. Both the CPU and GPU can be treated as devices in this case. The CPU simply runs threads in parallel to emulate the processing elements of GPU. Once a command has completed, a completion message is

passed to the host node. See Fig 5 for an overview of the SnuCL architecture.

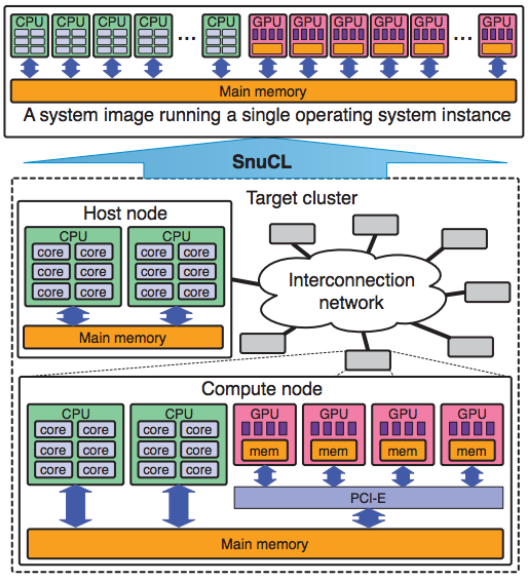

Fig. 5
SnuCL Architecture Overview

For scheduling, the SnuCL system takes special consideration for data locality. That is, for new tasks, the SnuCL runtime selects the source device that will incur the minimum copying overhead. Therefore commands that reuse data from previous commands will be assigned in such a way that the data does not need to be recopied.

Because SnuCL abstracts MPI communication, OpenCL programmers do not need to modify programming styles to utilize the SnuCL system, and OpenCL applications already written for single-node execution can be run on a distributed cluster without any modification.

E. libWater

The libWater [Main6] system is also based on OpenCL. This allows the system to support accelerators from several vendors, including Adapteva, Altera, AMD, IBM, Intel, and Nvidia. The libWater system is very similar to SnuCL, but with a few key differences.

Like SnuCL, the goal of libWater is to shift OpenCL programming from single-node execution to multi-node execution. However, as discussed before, performing this shift using MPI + OpenCL is error-prone and tedious. Therefore libWater aims to abstract the underlying distributed architecture such that remote devices can be used in the same way as local devices.

libWater is implemented as a C/C++ library-based extension to OpenCL. Because the main focuses of libWater are simplicity and productivity, the boilerplate OpenCL concepts of platform, device, queue, and context are abstracted into the libWater device construct. To correctly assign the device construct without the OpenCL boilerplate, libWater implements an original device query language (DQL) for selecting appropriate devices. That is, instead of using multiple OpenCL functions to select devices, they can be selected using statements like:

- "SELECT ALL WHERE (type = gpu AND vendor = nvidia)"
- "SELECT POS 1 FROM NODE 1 WHERE global_memory > 1024MB"

In SnuCL an entire cluster node is reserved as the host node for scheduling purposes, as well as one CPU core on each compute node. In contrast, libWater allows all nodes to act as compute nodes, including the host node, and does not reserve CPU cores on compute nodes for scheduling. See Fig 6 for an overview of the libWater architecture.

Because of this, another goal of libWater is for the runtime system to have very low resource utilization. CPU cycles dedicated to communication, scheduling, and other libWater runtime overheads directly compete with kernels executing on the CPU.

Another interesting contribution of the libWater system is the Dynamic Collective Replacement (DCR) optimization. libWater relies on underlying MPI calls for node-to-node communication. These communications, specifically individual send and receive requests, can significantly affect overall performance. However, most large computing clusters have specialized hardware to handle collective operations, such as scatter, gather, and broadcast. The goal of the DCR optimization algorithm is to replace multiple send and receive messages with a collective operation. They show that, under certain restrictions, the DCR optimiza-

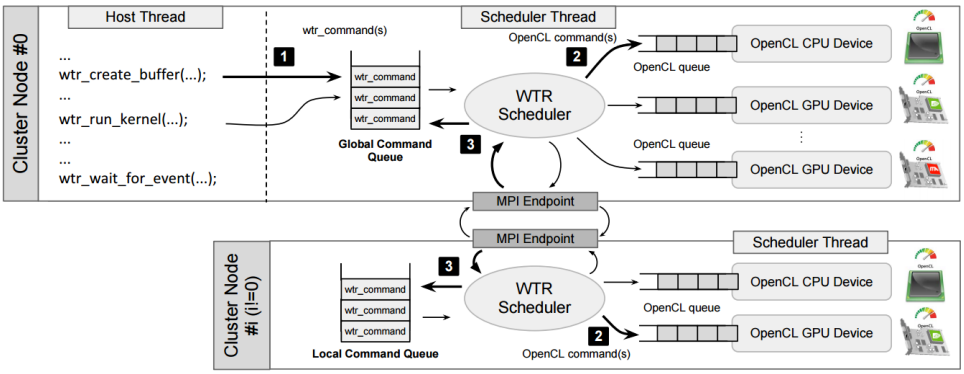

Fig. 6
LIBWATER'S DISTRIBUTED RUNTIME SYSTEM ARCHITECTURE.

tion can greatly improve performance. Therefore, they apply the optimization when these conditions are met.

Overall, the high level of abstraction of MPI calls and OpenCL boilerplate allow libWater to reduce the required amount of host code by a factor of two compared to an MPI + OpenCL implementation.

F. MT-MPI

MT-MPI [Main7] aims to extend the MPI + OpenMP approach to distributed heterogeneous computing. The other surveyed approaches require a move away from OpenMP to accelerator device back-ends like OpenCL and CUDA. However, MT-MPI is specifically designed to target Intel Xeon Phi devices, one of the few accelerators or co-processors that can be programmed with OpenMP, although an Intel OpenMP implementation is required.

However, even with the classic MPI + OpenMP approach, when the Intel Xeon Phi is used as a co-processor, certain inefficiencies arise, specifically the problem of idle threads previously discussed.

The authors modify the MPICH implementation of MPI and the Intel OpenMP runtime with the goal of eliminating the problem of idle threads. They also aim to parallelize MPI data copy operations.

To deal with idle threads, the OpenMP runtime exposes information about idle threads to the MPI runtime. This allows the MPI runtime to utilize these idle threads for communicating. The OpenMP threads are considered idle under two scenarios:

- when waiting for other threads at a barrier
- when waiting to enter a critical section

Once idle threads are identified by the OpenMP runtime and communicated to the MPI runtime, they can be temporarily repurposed to speed up data copies. However, the overhead of utilizing the idle threads outweigh the benefits for small messages, or for large messages with few idle threads. Therefore, the idle threads are only repurposed for large messages where a large number of idle threads are available. Otherwise MT-MPI falls back onto the sequential algorithm.

G. DART-CUDA

Like Phalanx, DART-CUDA [Main8] also moves away from the MPI paradigm, and is implemented using a PGAS runtime for communication.

DART-CUDA is based on the Dash Project, which, like Phalanx, is a C++ template library. However, instead of GASNet, the Dash project uses DART-SYSV and System V threads to implement the underlying communication.

Much of the DART-CUDA paper covers in detail the Dash Project and Dart API, which are beyond the scope of this survey.

However, as one of the newer papers surveyed, the DART-CUDA implementation may signal a potential shift toward PGAS implementations for heterogeneous clusters. Specifically, they mention that with PGAS systems, many challenges like in-

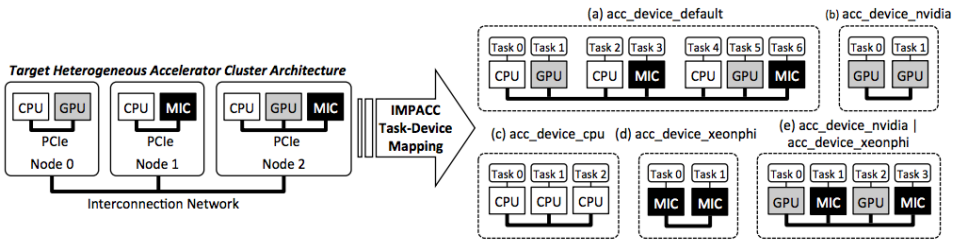

Fig. 7
AUTOMATIC TASK-DEVICE MAPPING IN A HETEROGENEOUS CLUSTER WITH DIFFERENT NODES.

tegration of separate memory spaces into a global address space, global access of objects, and locality of reference are all shifted to the compiler. This aids in programmer productivity and abstracts away many hardware details.

H. IMPACC

The final and newest paper surveyed introduces IMPACC [Main9] (Integrated Message Passing ACCelerator programming), which is an extension of the MPI + OpenACC approach to heterogeneous distributed computing.

As previously discussed, OpenACC provides a high-level alternative to CUDA and OpenCL for accelerator programming. IMPACC is implemented using the OpenARC compiler. OpenARC is a source-to-source compiler that translates and input program into the appropriate device language. That is, it generates CUDA code when targeting Nvidia devices, OpenMP code when targeting CPU multi-processors, and OpenCL for other processors like FPGAs and Xeon Phis. This allows IMPACC to be applied over the widest scope of devices from all programming approaches surveyed.

Most of the effort of the IMPACC programming system is spent addressing inefficiencies in the standard MPI + OpenACC model. They address the problems by tightly integrating the MPI and OpenACC calls, and by performing several optimizations. IMPACC also proposes several extensions to OpenACC to aid in heterogeneous distributed execution.

One advantage of IMPACC is its ability to execute in a heterogeneous cluster with not only CPUs and accelerators, but also different types of accelerators. This is shown in Fig 7.

III. MAIN THEMES

In this section we give an overview of the main themes realized from the survey of distributed heterogeneous programming approaches. While mostly features unique to each programming approach were highlighted in Section II, many of the programming approaches overlap in their implementations. Also, many of the later implementations were inspired by and reference earlier implementations. For example, the IMPACC system is directly built off of components of the SnuCL system.

A. Platform Model

The platform model refers the hardware architecture targeted by the programming models. While all programming approaches target a heterogeneous distributed system, some are restricted to specific devices or specific architecture configurations, determined by the device-level back-end used in the implementation.

(i) **CUDA**: Many of the programming systems only target GPUs for accelerators. This is not unreasonable, as GPUs are by far the most popular and widespread accelerator used in heterogeneous computing. The OmpSs, Phalanx, and DART-CUDA all only target Nvidia CUDA devices. While this limits their applicability, CUDA is one of the oldest and most well-supported accelerator APIs, and Nvidia GPUs are among the most popular for GPGPU programming. The IMPACC system can target CUDA as well as OpenCL.

(ii) **OpenCL**: Most of the programming systems surveyed, including StarPU, SnuCL, libWater, and IMPACC, target OpenCL as a device language. This is a natural choice, as it allows the system to to execute across a diverse set of accelerators and co-processors, including GPUs, Xeon Phis, and FPGAs.

(iii) **OpenMP**: While most systems support OpenMP as the back-end for multi-core CPU processing, only MT-MPI uses OpenMP as the back-end for its heterogeneous component, the Intel Xeon Phi. This severely limits the applicability of MT-MPI. However, the most recent release of the Xeon Phi, the Knights Landing (KNL), no longer supports OpenCL. This leaves OpenMP, and systems like MT-MPI, as one of the few options for high-level heterogeneous programming of Xeon Phi clusters.

B. Programming Model

The programming model refers to the the programming style exposed to programmers by each system or model, and the types of abstractions implemented over the low level communications and low level device languages.

Most of the programming systems move away from the hybrid MPI + X approach and adopt a unified programming approach. StarPU, OmpSs, Phalanx, libWater, and SnuCL all adopt a unified approach. With a unified approach, the programmer only needs to reference one set of API calls to write heterogeneous distributed programs. This can greatly simplify programming efforts, and avoids potential clashes or unexpected behavior when mixing two APIs. Also, a unified API can abstract away much of the hardware details.

Two of the papers surveyed continue to use a hybrid approach, MT-MPI with an MPI + OpenMP, and IMPACC with MPI + OpenACC. There are two important factors that allow these systems to work well with the hybrid approach. First, both OpenMP and OpenACC are high-level directive based languages that already abstract away much of the underlying communication and hardware details. And second, both MT-MPI and IMPACC rely on a tightly integrated approach. MT-MPI modifies both the MPI and OpenMP runtimes to remove inefficiencies from the mixing of the APIs. Similarly, IMPACC extends OpenACC to tightly integrate with MPI calls. While still technically a hybrid approach, this tight integration of the two programming APIs makes these approaches similar to the unified approaches.

Within the unified and hybrid approaches, the APIs can be broadly categorized into three types.

- **Directive Based**: OmpSs, IMPACC, MT-MPI
- **C/C++ Libraries**: Phalanx, DART-CUDA
- **Modifying Underlying Runtimes**: SnuCL, libWater, MT-MPI, IMPACC

C. Communication and Memory

Several different strategies are implemented for communicating between different nodes and within nodes in a distributed system. Also, how the distributed system memory is abstracted to the programmer varies between the surveyed programming approaches.

The two biggest distinctions in communication and memory management are between the classic MPI approach, and the new PGAS approach. With the MPI approach, each process has a distinct address space, and the programmer explicitly manages communication of data between nodes by passing messages, in the form of sends and receives. In the PGAS model, nodes share a global address space, and data is communicated using what is known as "active messages" in the form of get and put.

- **MPI** Unsurprisingly, most of the programming approaches adopt MPI as the underlying method of communication. StarPU, OmpSs, libWater, SnuCL, MT-MPI, and IMPACC all rely on MPI for node-to-node communication.
- **PGAS** Phalanx and DART-CUDA are the only two approaches surveyed that rely on the PGAS model for communication and memory management. Although they are in the minority here, many PGAS languages and systems have recently risen in popularity as MPI-alternatives.

IV. Conclusion

As computing clusters and large supercomputers continue to shift from homogeneous architectures toward heterogeneity, it is essential to develop and improve programming strategies for these systems. Classic hybrid MPI + X approaches offer one option for programming these heterogeneous systems. However, these models introduce many inefficiencies, both in computational performance and in programmer productivity.

The programming models and systems surveyed here offer high-level, unified, or tightly integrated alternatives to the MPI + X approach. The abstractions and optimizations implemented by these systems are shown to improve computational performance and programmer productivity.

In addition to the approaches surveyed here, several other classes of approaches have recently become popular for heterogeneous distributed computing. Map-reduce frameworks like MapReduce, Sparc, and Hadoop offer another alternative to MPI for GPU-cluster computing. Also, the recent release of the OpenMP 4.0 standard includes support for offloading to GPU accelerators. This may result in a shift back to the class MPI + OpenMP approach.

In conclusion, due to the increasing number of specialized hardware devices, the growing trend of heterogeneous computing is likely to heavily influence the future of high-performance and distributed computing. These systems will rely on the continued development of heterogeneous distributed programming systems and models like the ones surveyed here.

Main References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[2] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault, "Starpu-mpi: Task programming over clusters of machines enhanced with accelerators," in *EuroMPI*. Springer, 2012, pp. 298–299.

[3] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta, "Productive programming of gpu clusters with ompss," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 557–568.

[4] Michael Garland, Manjunath Kudlur, and Yili Zheng, "Designing a unified programming model for heterogeneous machines," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 2012, pp. 1–11.

[5] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 341–352.

[6] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer, "Libwater: heterogeneous distributed computing made easy," in *Proceedings of the 27th international ACM conference on International conference on Supercomputing*. ACM, 2013, pp. 161–172.

[7] Min Si, Antonio J Peña, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa, "Mt-mpi: Multithreaded mpi for many-core environments," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 125–134.

[8] Lei Zhou and Karl Fürlinger, "Dart-cuda: A pgas runtime system for multi-gpu systems," in *Parallel and Distributed Computing (ISPDC), 2015 14th International Symposium on*. IEEE, 2015, pp. 110–119.

[9] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter, "Impacc: A tightly integrated mpi+ openacc framework exploiting shared memory parallelism," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 189–201.

Other References

[1] Leonardo Dagum and Ramesh Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[2] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008.

[3] John E Stone, David Gohara, and Guochun Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[4] "The openacc application programming interface," 2013.

[5] Message P Forum, "Mpi: A message-passing interface standard," Tech. Rep., Knoxville, TN, USA, 1994.

[6] Franck Cappello and Daniel Etiemble, "Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, pp. 12–12.

[7] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda, "Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, pp. 257, 2011.

[8] Tsuyoshi Hamada and Keigo Nitadori, "190 tflops astrophysical n-body simulation on a cluster of gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–9.

[9] NP Karunadasa and DN Ranasinghe, "Accelerating high performance applications with cuda and mpi," in *Industrial and Information Systems (ICIIS), 2009 International Conference on*. IEEE, 2009, pp. 331–336.

[10] Dan Bonachea, "Gasnet specification, v1. 1," 2002.