Unified Tasks and Conduits for Programming on Heterogeneous Computing Platforms

A Dissertation Presented
by

Chao Liu

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

December 2017

To my family.

Contents

List of Figures

List of Tables

List of Acronyms

APGAS Asynchronous Partitioned Global Address Space

ASIC Application Specific Integrated Circuit

DAG Directed Acyclic Graph

FPGA Field Programmable Gate Array

GPU Graphic Processing Unit

ISA Instruction Set Architecture

LOC Lines of Codes

PGAS Partitioned Global Address Space

PVTOL Parallel Vector Tile Optimizing Library

MPI Message Passing Interface

NUMA Non-uniform Memory Access

RMA Remote Memory Access

TnC Tasks and Conduits

UTC Unified Tasks and Conduits

Acknowledgments

Here I wish to thank those who have helped and supported me during the process of my Ph.D. study. I would first like to thank my advisor Prof. Miriam Leeser who I have been working with for the last four years. She led me to this research project and gave me precious advice all the way through my research work. She is very knowledgeable and never holds back anything to help me. Whenever I have uncertainty or problems with my work she can always come up with insightful ideas and give me inspirations. She is very responsible and careful. I can remember the every tiny corrections she have made for all my paper works. I am thankful for all the time Prof. Leeser dedicated to my project. This work would not be possible without her incredible support and guidance.

I would like to thank my other committee members Prof. Ningfang Mi and Prof. Stefano Basagni for their direction and expertise. I am very appreciate for their valuable suggestions during my proposal and careful revision of my dissertation work.

I want to say thank you to my colleagues and friends for their encouragement and help: Xin Fang, Kai Huang, Ben Drozdenko, Mahsa Bayati, Janki Bhiman, Jieming Xu and all my friends. Because of them I am enjoying my Ph.D. study and I will never forget the every pleasant moments we were together. Thanks also goes to all of the professors and teachers I have had that have helped me complete my study.

Most important, a very special thanks to my wife, my parents and my sisters. Their endless love and constant support and encouragement get me to this point. I feel so lucky to have them beside me. Thanks for everything.

Abstract of the Dissertation

Unified Tasks and Conduits for Programming on Heterogeneous Computing Platforms

by

Chao Liu

Doctor of Philosophy in Computer Engineering

Northeastern University, December 2017

Dr. Miriam Leeser, Advisor

Computing platforms for high performance and parallel applications have changed from traditional Central Processing Units (CPUs) to hybrid systems which combine CPUs with accelerators such as Graphics Processing Units (GPUs), Intel Xeon Phi, etc. These developments bring more and more challenges to application developers, especially to maintain a high performance application across various platforms. Traditional parallel programming methods are usually low-level. They focus more on data distribution and communications, aiming to generate high performance and scalable applications. The programs are usually closely related to the underlying platform. Once the platform changes or programs are ported to different environments, lots of effort is needed to modify or reprogram.

To reduce development effort and improve portability of applications we need a programming method that can hide low-level platform hardware features to ease the programming of parallel applications as well as maintain good performance. In this research, we propose a lightweight and flexible parallel programming framework, Unified Tasks and Conduits (UTC), for heterogeneous computing platforms. In this framework, we provide high level program components, tasks and conduits, for a user to easily construct parallel applications. In a program, computational workloads are abstracted as task objects and different tasks make use of conduit objects for communication. Multiple tasks can run in parallel on different devices and each task can launch a group of threads for execution. In this way, we can separate an applications' high-level structure from low-level task implementations. When porting such a parallel application to utilize different computing resources on different platforms, the applications' main structure can remain unchanged and only adopt appropriate task implementations, easing the development effort. Also, the explicit task components can

easily implement task and pipeline parallelism. In addition, the multiple threads of each task can efficiently implement data parallelism as well as overlapping computation and communication.

We have implemented a runtime system prototype of the Tasks and Conduits framework on a cluster platform, supporting the use of multicore CPUs and GPUs for task execution. To facilitate muti-threaded tasks, we implement a task based global shared data object to allow a task to create threads across multiple nodes and share data sets through one-sided remote memory access mechanism. For GPU tasks, we provide concise interfaces for users to choose proper types of memory for host/device data transfer. To demonstrate and analyze our framework, we have adapted a set of benchmark applications to our framework. The experiments on real clusters show that applications with our framework have similar or better performance than traditional parallel implementations such as OpenMP or MPI. Also we are able to make use of GPUs on the platform for acceleration through GPU tasks. Base on our high level tasks and conduits design, we can maintain a well organized program structure for improved potability and maintainability.

Chapter 1

Introduction

In the long era of single core processors, the main improvements derived from higher frequency and more complex micro-architecture. However, after encountering physical hardware limitations and energy consumption problems, multicore and manycore technology has become the main path to providing continuing compute ability improvement. With the massive use of multicore processors and manycore accelerators, we are now in the parallel era.

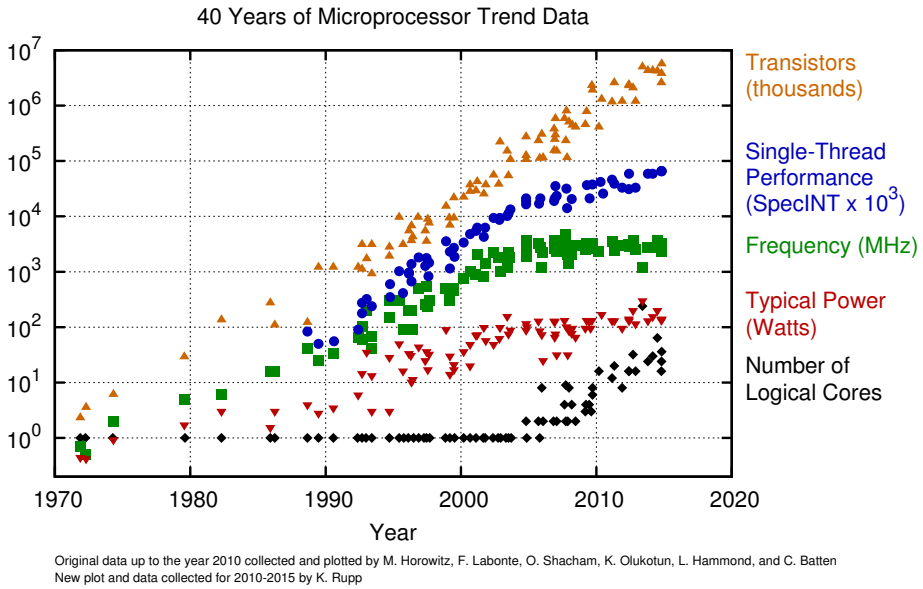


Figure 1.1: Processor Development Trend [1]

In the single-core era, applications, which are sequential programs, can benefit from every

new generation of processor without changing or reimplementing the applications. However, in the parallel era, developing applications with parallel programming becomes the critical method to benefit from hardware improvement. Parallel applications allow a user to take advantage of underlying parallel processing resources; they also bring more challenges to the application developer. Fast-changing and complicated hardware platforms compel the applications to be modified to adapt to the platforms' new features, thus application portability and maintenance becomes a major problem. Furthermore, the popularity of dedicated processors or accelerators such as GPUs, FPGAs and other ASIC chips exacerbates this problem. To utilize new devices, lots of development effort is required to manage these devices and reprogram the application with specific programming methods. For GPUs for example, a user needs to manually manage the usage of each GPU in the application, and design and implement the parallel algorithms through GPU supported programming methods. To produce high-performance parallel applications, more and more device architectures or hardware features are exposed to the user. User may need to be familiar with GPU architecture to implement high performing GPU programs, or may need to learn hardware logic in order to program FPGAs. This brings more burden to application development, especially for domain experts who are not familiar with computer hardware. Developing applications and running on various ever-changing platforms with as little modification of the applications as possible is an important goal in the high-performance parallel computing area.

Accompanying the advancement of parallel computing hardware, there are also various parallel programming models or methods proposed to facilitate parallel application development. For single node systems, shared memory programming methods such as threads [4] or OpenMP [5], are widely adopted. For cluster systems, message passing model (MPI) [6] is the most popular programming method. With the development of web technology and the emergence of cloud computing, programming methods on large distributed systems, such as Mapreduce [7] and Scala [8] are more and more being used in web service applications. These programming methods are either implemented through a new language or a language directive plus compiler approach, or they work using a runtime library and supply APIs for use. Compared to new languages and automatic parallelization through a compiler, library based approaches are more flexible and often deliver better performance. They require the user to express parallelism explicitly in the programs, thus also increasing programming effort. However, traditional parallel programming methods, especially for high-performance computing, are usually low-level programming. They focus more on data distribution and communication aspects, aiming to generate high performance and scalable applications. The programs usually are closely related to the underlying platform. Once the platform

changes or programs are ported to different environments, lots of effort is needed to modify or reprogram. Using accelerators requires even more effort to manage the devices, combining different programming models or methods together in parallel applications. To solve these problems, we need a programming method that can hide low-level platform hardware features to ease the parallel application development and improve portability and maintenance. This is where the tasks and conduits model (TNC) [9] is proposed. TNC provides simple and concise high-level concepts (tasks and conduits) to abstract the computation and communication workloads. An application composed of task and conduit components is intended to run across a variety of platforms with little modification.

The original TNC implementation mainly targeted signal processing applications. The task and conduit components were adapted to the features of signal processing applications, limiting the applicability of TNC. Thus it was not easy to extend to support new hardware features. To this end, we propose a lightweight and flexible framework based on the TNC model. In this framework, parallel application development is divided into two levels: the higher main structure level, which is constructed with task and conduit components; and the lower task implementation level, which uses platform preferred programming methods to realize customized computation. With this framework, we can explore the balance between program portability, adaptability, and performance. When porting applications to different platforms, the program main structures stay unchanged and only the task implementations change, reducing development effort. Task implementation is still low level and platform related, but with a well-defined interface and glue code, common computations can be implemented as task templates or libraries for easy reuse. Furthermore, by extending and integrating different target computing devices into the framework, a user can continue benefiting from new devices with less effort, mainly focusing on application design and algorithm implementation.

The primary contributions of this dissertation work are:

- **Unified Task and Conduit framework**: Based on the TNC model, we introduce a lightweight and flexible parallel programming framework, UTC, which aims to help the user develop parallel applications on heterogeneous computing platforms with improved portability.

- **Framework runtime system**: To support the UTC framework, we design and implement a runtime system on a cluster platform. Through a library based approach and the C++ language, we create Task and Conduit classes, together with other utility functions, to enable a user to easily define and instantiate tasks and conduits to construct parallel applications.

- **Multi-threaded task execution**: We design and implement task execution with multiple threads. This enables users to implement tasks as multi-threaded parallel programs to explore data parallelism easily, utilizing multicore processors for execution.

- **Task-based global shared data object**: We implement a task-based global shared data object, which enables multi-threaded task programs to run on multiple nodes, sharing data sets through global shared data objects and accessing data with one-sided remote memory access (RMA).

- **GPU task support**: We extend the framework runtime and implement utilities and glue functions to control and manage GPUs, supporting users to program GPU kernels with CUDA and create GPU tasks to make use of GPUs for acceleration. Through CUDA context and CUDA stream binding, users can utilize a single GPU or GPU clusters effectively. Further, we provided uniform and simple interfaces to facilitate users choosing and exploring different types of memory for host/device data transfer when implementing GPU tasks.

- **Use of framework for application development**: We ported a set of benchmark applications with the UTC framework to analyze and demonstrate the usability as well as the performance of the framework and its implementation on a heterogeneous cluster platform.

The remainder of this dissertation is organized as follows: Chapter 2 discusses some background and related work about parallel computing platform and programming models and methods. Then Chapter 3 details the framework design and framework implementation on heterogeneous cluster platforms. We discuss the GPU support in our framework runtime as well as the task based global shared memory design and implementation. Chapter 4 shows the development of benchmark applications and experiments and analysis of running these applications on real clusters. Chapter 5 gives a summary and some future work.

Chapter 2

Background

2.1 Modern Parallel Computing Platforms

With decades of development of computer architecture and hardware technology, multicore and manycore processors have become prevalent. Also, coprocessors or accelerators, such as Graphics Processing Units (GPUs), which in the past were used in limited and specific applications, are now more and more being used for general computing, due to improvements in software and hardware techniques [10]. The fast changing and wide use of these new processors has led to the rapid improvement of parallel computing platforms.

2.1.1 Multicore and Manycore Processors

After encountering physical limitations in designing single core processors, computer hardware vendors moved to multicore technology. The basic multicore CPU architecture is shown in Figure 2.1. In a multicore processor, each core has its own instruction fetch unit and execution unit. L1 cache and L2 cache usually are private for each core and fully coherent, while all the cores share the L3 cache. All the cores share a common system memory and can execute instructions in parallel. Today, from high-end server computers to personal desktops or mobile devices, all are using multicore processors to provide powerful processing capabilities. Current development trends show that the number of CPU cores will keep increasing. The latest AMD Zen architecture CPU will comprise up to 32 cores, and can execute 64 logical threads in parallel with hyper-threading [11].

Besides traditional multicore CPUs, coprocessor and accelerator technology also keeps improving, especially the prevalence of using GPUs for general purpose computing. There are many

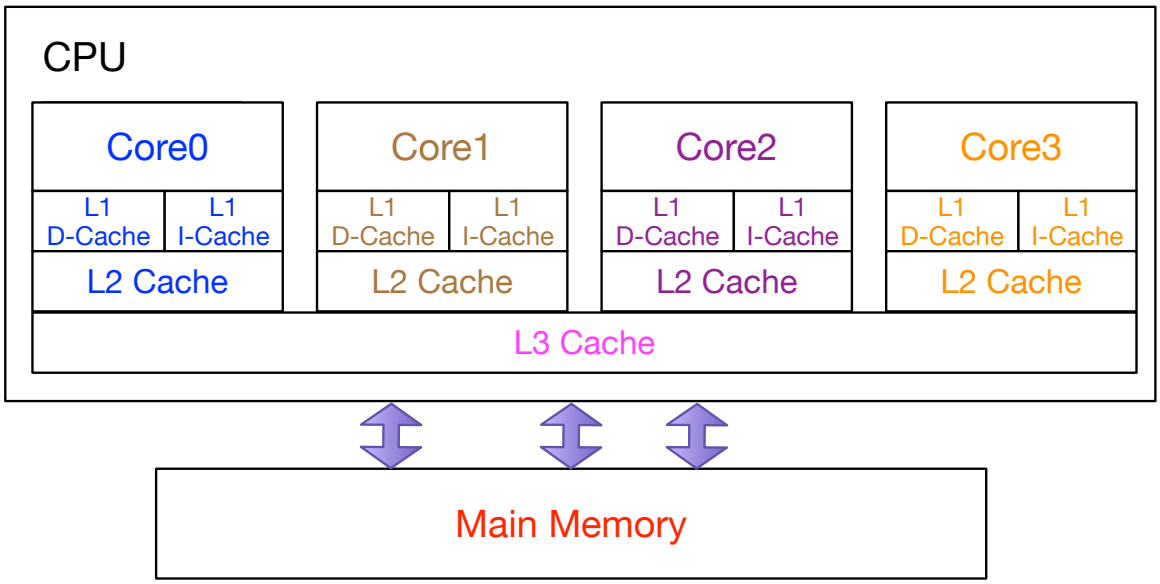


Figure 2.1: Simplified Multicore Processor

differences between CPUs and GPUs. A GPU has many more processing cores (called streaming processors-SPs) than a CPU. For example, the NVIDIA Pascal GPU (GP100) contains up to 3840 SPs. However, the logic complexity of each SP is much simpler than CPU cores and operates at a lower frequency.

The general architecture of NVIDIA GP100 GPU is shown in Figure 2.2. As shown, it contains 6 Graphics Processing Clusters (GPCs), and each GPC has 10 Streaming Multiprocessors (SMs). In each SM, there are 64 SPs, or a total of $64 \times 10 \times 6 = 3840$ cores. All the GPU cores share a 4MB L2 cache and can access up to 16GB off-chip global memory. For each SM, 64 SPs share a 64KB L1 cache and 64KB local shared memory which can be explicitly allocated for access in users' programs. More detailed features of this GPU architecture are described in [2]. Through thousands of processing cores, GPUs can perform massively parallel computation and provide dramatic acceleration. One GP100 GPU can achieve 10 TFlops single precision peak performance.

Another popular recent accelerator is the Intel Xeon Phi coprocessor [12]. In 2006 Intel initiated its manycore processor development with the Larrabee project [13]. Inheriting from Larrabee, Intel announced its development of Many Integrated Core (MIC) architectures, based on which Xeon Phi processors are developed. The first generation of Xeon Phi processor was revealed in 2012, named the Knights Corner (KNC) processor. Now the second generation of Xeon Phi processor, named as Knights Landing (KNL), is available. The micro architecture of KNL is show in Figure 2.3.

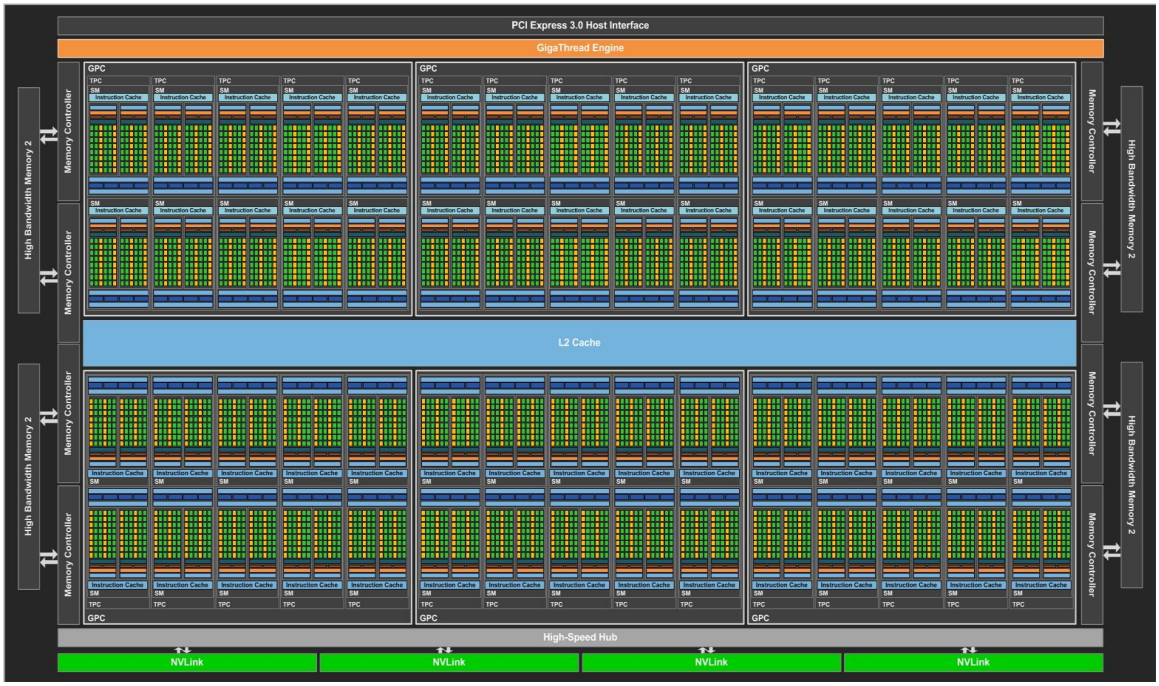


Figure 2.2: GP100 GPU Architecture [2]

It has up to 72 cores which are divided into 32 Tiles connected by a 2D mesh network. Each Tile includes two cores and shares a 1MB L2 cache. All 72 cores share 16 GB on-package memory and also can access very large external system memory through 6 DDR4 channels. Every core can have up to 4 threads active simultaneously, running a total of 288 threads in parallel and providing 6 TFlops single precision peak performance. Compared to GPU, a key feature of the Xeon Phi processor is that it aims to provide a general purpose programming environment similar to the traditional CPU environment. The Xeon Phi processor is ISA compatible with the x86 Intel Xeon CPUs and supports the common x86 memory order model, therefore it is capable of running applications written in traditional programming languages such as Fortran or C. Because of its general purpose features, the Xeon Phi processor can run general operating systems. It can be either used as a PCI-e device in conjunction with host Xeon CPUs, similar to GPU, or be used as a stand-alone general purpose CPU.

In addition to making use of GPUs or Xeon Phi coprocessors to accelerate general purpose computing, more and more dedicatedly designed hardware is used for domain specific applications. For example, in the data center and deep learning areas, FPGAs are recently adopted to increase data processing speed as well as reduce power consumption [14, 15].

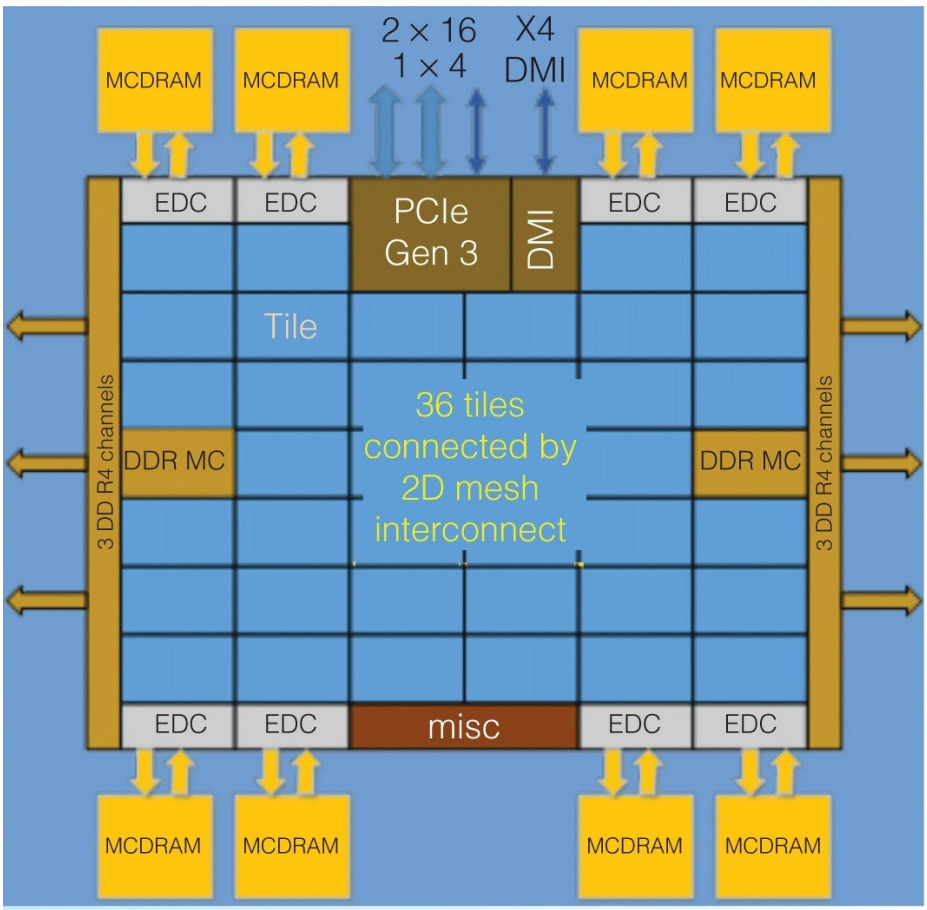


Figure 2.3: Intel Xeon Phi Knights Landing Micro Architecture [3]

2.1.2 Heterogeneous Computing Platforms

The use of various processors in a system converts the traditional homogeneous computing platform to a novel hybrid heterogeneous system. A typical structure of parallel computing platforms is show as Figure 2.4.

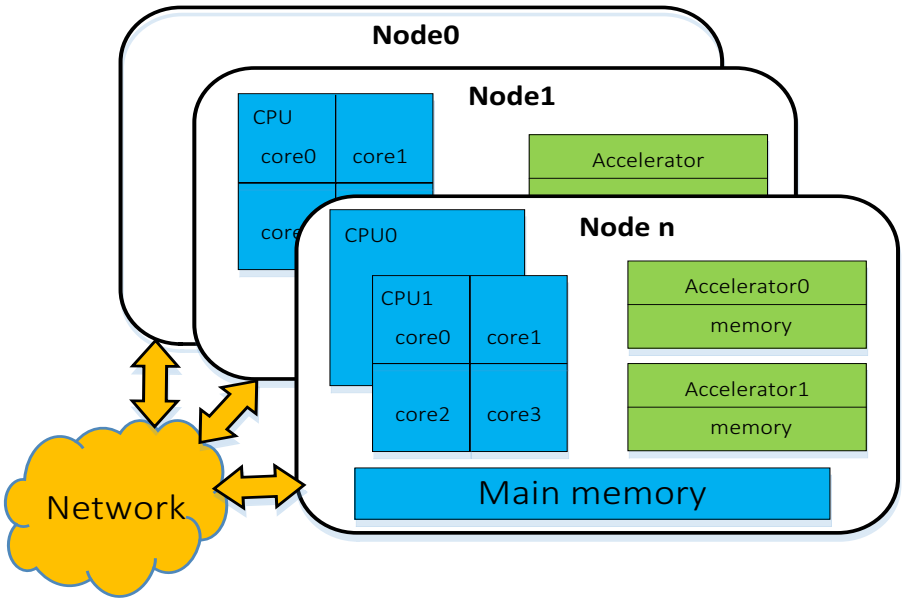


Figure 2.4: Parallel Computing Platform Architecture

In this platform, a compute node may have one or more traditional multicore CPUs that run the operating system and execute I/O, network and other services, as well as have direct access to main memory. In addition, there may be one or several accelerators in a compute node which can execute workloads when a control program running on the CPU assigns work to them. Usually, accelerators have their own device memory and cannot access main memory directly. Several compute nodes are connected through a high-speed network fabric, forming a cluster. This platform contains lots of multicore and manycore processors, providing plenty of parallel computing resources. On the other hand, the combination of different kinds of processors and complicated memory systems bring more and more challenges to developing and maintaining high-performance parallel applications on such platforms.

2.2 Programming Methods for Parallel Platforms

Targeting parallel computing platforms, there are many parallel programming models and methods that have been proposed to assist users to design and develop applications that explore the computing ability of these platforms.

2.2.1 Traditional Parallel Programming Models

Based on the memory accessing features in parallel computing systems, there are two primary parallel programming models, shown in Figure 2.5: Shared memory and distributed memory.

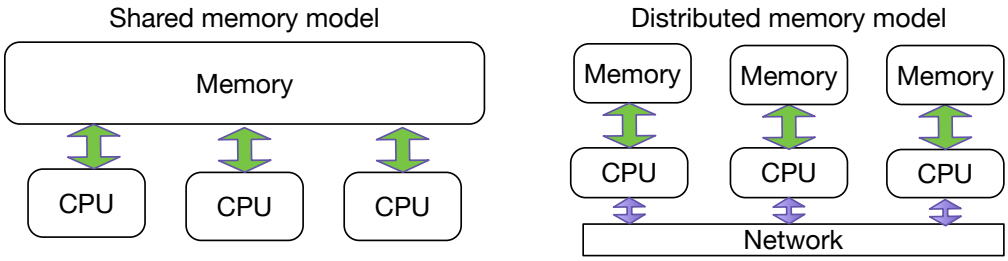


Figure 2.5: Two Basic Parallel Programming Models

Shared Memory Programming Model

With the shared memory model, parallel jobs run on different cores but share the same memory address space. Parallel jobs can share information with each other directly through shared memory, and data communications between parallel jobs occurs implicitly. Therefore there is less burden for the user to develop parallel applications with this model. The user does need to care about the consistency of shared data and apply synchronizations properly. However, shared memory programming models are only suited to single compute nodes, so a user cannot develop applications to run on multiple nodes.

The most common and widely used parallel programming methods with this model are POSIX Threads (Pthreads) [4] and OpenMP [5]. Pthreads is a library-based programming method. It standardizes and implements a set of functions for users to create and manage multiple threads that run concurrently in the system. With Pthreads, users have full control of every thread, and users need

to manage threads properly to avoid data races and deadlock. It is a challenge for the user to handle parallel programs when the number of threads gets very large.

OpenMP, which is implemented based on Pthreads or other threading libraries, uses a set of compiler directives to assist the compiler in generating parallel programs automatically. It allows users to insert compiler directives in their C or Fortran programs. And with the help of the compiler and runtime libraries, groups of threads will be generated for parallel execution. Unlike Pthreads, threads in OpenMP programs are spawned and managed implicitly, without the users' attention, so it is easier for users to develop parallel programs with OpenMP, especially to implement data parallelism for loop structures.

Distributed Memory Programming Model

For clusters of compute nodes connected by a network, each node has its own memory and they are physically separated. The distributed memory programming model assists parallel application development. On such a platform Message Passing Interface (MPI) is the de facto standard distributed memory programming model. The most critical part of MPI is that it provides a set of library routines to implement efficient data communications between parallel jobs across a network. With MPI, users can initialize a group of processes for parallel execution. These processes can either run on a single node or on a cluster of nodes. Each process has its own memory address space for data access. Different processes may complete data communication through invoking MPI send/receive functions no matter whether they are on the same node or not. Through MPI, users can develop parallel applications running on a large number of compute nodes. However, the performance of parallel programs usually highly depends on the communication pattern in the programs as well as the implementation of network data transfer in MPI runtime libraries. Also, because MPI programs use an explicit two-sided message passing pattern to implement parallel algorithms, developers need to coordinate the data distribution and movement explicitly in the program. This is harder to program than the shared memory programming model. Widely used MPI implementations include MPICH [16], MVAPICH [17] and OpenMPI [18].

Partitioned Global Address Model

Benefiting from improvements in network hardware and software, especially Remote Memory Access (RMA) techniques, it is possible to virtualize remote memory access operations as local memory access, using one-sided communication instead of two-sided communication.

Therefore the Partitioned Global Address Model (PGAS) [19] is proposed. In PGAS, multiple pieces of distributed memory are virtually mapped to a single memory address space and shared by all the processing cores as shown in Figure 2.6. Each core can access data in the virtual global space directly no matter where in memory they are. The PGAS model extends the traditional shared memory concept to a distributed memory environment, allowing a group of processes to share a virtual global memory space across multiple nodes. It makes use of one-sided communication mechanisms and enables users to access remote data from different processes with the same address pointers.

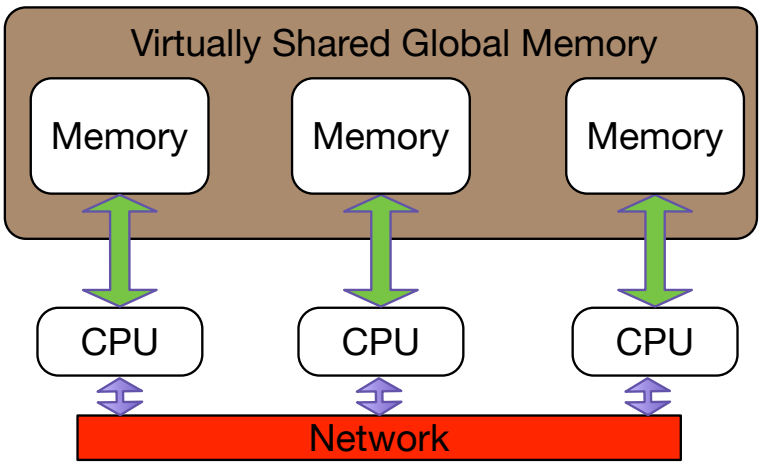


Figure 2.6: PGAS Model

Currently, the PGAS model is implemented in two manners. One is language-based implementations, such as Unified Parallel C [20], Titanium [21] and Co-Array Fortran [22]. These programming languages usually extend C or Fortran and use compiler and runtime libraries to realize the PGAS memory model implicitly. The other direction is library-based, such as OpenSHMEM [23]. Similar to MPI, it provides a set of libraries routines to allow users to perform one-sided and direct data accessing operations.

2.2.2 Programming Languages and Methods for Accelerators

The above parallel programming methods or languages all target CPUs. Due to the architecture differences between CPUs and accelerators, specific programming methods are introduced to develop parallel programs for accelerators.

CUDA [24] is one of the most popular programming languages for programming GPUs. CUDA was developed by NVIDIA for implementing general purpose computing on their GPU products. By extending the C/C++ grammar and adding a series of functions, CUDA enables users to initialize GPU devices, transfer data between main memory and GPU memory and launch kernel programs on GPUs. A GPU kernel program which utilizes a GPU for execution is composed of tens of thousands of GPU threads. Each GPU thread runs on a GPU core. In order to implement parallel programs, CUDA introduces a two levels of threads and memory hierarchy pattern to map the massively parallel GPU program threads to a GPU architecture effectively. The unique features of GPU architectures make programming GPUs much different from programming CPUs, and lots of algorithms need to be redesigned and reimplemented. To design a high quality parallel application on GPUs also requires developers to be well informed about GPU architecture. All these increase the burden and difficulty of developing parallel applications on GPUs.

OpenCL [25] is another popular programming method for GPU. Unlike CUDA which only supports NVIDIA GPUs, OpenCL is an open standard for programming manycore architectures. Both AMD GPUs and NVIDIA GPUs provide implementation support for OpenCL. AMD also supports running OpenCL programs on their multicore CPUs. OpenCL has a similar threads/memory hierarchy pattern as CUDA, so parallel algorithms designed with them are similar. In addition to CUDA and OpenCL, OpenACC [26] is another programming language for GPUs. It adopts the concepts of OpenMP. By inserting compiler directives in sequential programs, the compiler will generate the parallel kernels for running on GPUs. This approach reduces the difficulty of programming on GPUs. However, the performances of programs highly depends on the compiler and is not always good quality.

2.2.3 Hybrid Parallel Programming

Different parallel programming models or methods have different features and may be preferable in different situations. Using a mixture of different programming models, also known as hybrid parallel programming [27], is being explored for better performance. For example, MPI is used for distributed memory systems. It provides well-defined communication methods for running parallel programs crossing multiple compute nodes. Meanwhile, it is easier and intuitive to use a shared memory parallel programming model on a single node with multicore CPUs. Thus “MPI + X” hybrid parallel programming methods are used to develop applications on multicore cluster platforms. The most common experience is mixing MPI and OpenMP [28]. For accelerators such

as GPUs, CUDA and OpenCL are usually used to develop programs to be run on single GPUs. To make use of multiple GPUs at the same time, other programming tools are needed. MPI + CUDA or Pthreads + CUDA are frequently used for programming GPU clusters.

2.3 Task Parallelism and the TNC Model

Traditional parallel programming models or methods are usually low level. They focus primarily on data distribution and are suitable for implementing data parallelism for a kernel function in an application. They lack the abstraction of task parallelism. For an application composed of several workload parts, it is hard to use these parallel programming models to express the applications using higher level components. Because of the low-level features of traditional programming methods, when changing from one platform to another, lots of modifications to the application are required to target a new platform. This increases the burden to develop and maintain parallel applications. Also using these low-level programming methods, users need to familiarize themselves with the features of each hardware platform to tune parallel applications. This makes it much harder for domain experts who lack hardware knowledge or are not familiar with novel processing devices to produce high quality parallel applications.

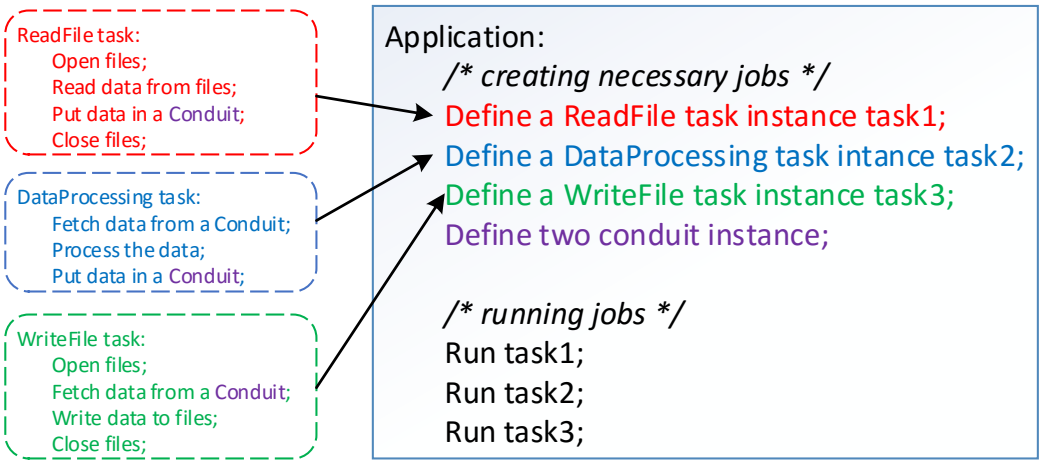


Figure 2.7: Application with TNC Model

The Tasks and Conduits (TNC) model provides well-defined concepts for implementing task parallelism in an application. TNC allows a user to express explicit task parallelism easily and naturally. Using the TNC model as a high-level abstraction for applications was first presented in [9]. To facility the development of signal processing applications, the authors provided a Parallel

Vector Tile Optimizing Library (PVTOL) to help developers create portable parallel programs on multicore platforms and the TNC model is used to design and implement applications. A typical application using the TNC model is shown in Figure 2.7. In this model, tasks are the abstraction of computation workloads; conduits are the abstraction of data transfer between different tasks. Different computationally intensive pieces of an application can be defined as several task objects and these tasks cooperate through conduits. Inside each task, detailed computational logic is implemented. In this way, an application's main structure can be expressed at the task level, which is independent of the task implementation and execution on a specific platform. The variety of features available in hardware can be hidden from software applications. This library is also extended to support running applications with GPUs [29]. However, PVTOL is mainly targeted at signal processing applications. The task implementation works in a strict SPMD pattern: each parallel job of the task works on its own data. This restricts the usage of this model. Also, the task and conduit components are not implemented as a single layer, making it hard to change and extend.

2.4 Related Work

As multicore and manycore processors become prevalent, there is growing research interest in parallel programming and developing parallel applications. Data parallelism and task parallelism are the two primary patterns to design and implement parallel applications. To implement data parallelism, OpenMP and MPI are still the most widely used programming methods in shared memory and distributed memory systems. Through compiler directives OpenMP provide an easy way to help users run programs with multiple threads for parallel processing on single node shared memory system. MPI, on the other hand, maintains a set of high performance communication methods to enable users to implement and run parallel programs on distributed cluster system. But the explicit message pattern and distributed memory requires more programming effort. Our framework implementation combines the shared memory feature and message passing feature together: a single task can be implemented as a multi-threaded parallel programs in which shared memory concepts are applied; different tasks communicate and move data through conduits explicitly which has the message passing feature.

The PGAS model introduces a distributed shared memory space to ease programming effort with a shared memory model and one-sided communications. Under the PGAS model, memory blocks on one node can be virtually shared with other nodes through one-sided remote memory access. This one-sided RMA communication gives users more flexibility to move data sets between parallel

units and overlap communication with computation. Some PGAS implementations are built upon GASNet [30]. There is also research for realizing PGAS based on MPI one-sided communications [31, 32, 33, 34]. In this work we also utilize MPI to carry out the one-sided RMA operation. But unlike those works which aimed to build up a system wide PGAS memory model, we make use one-sided RMA to realize distributed shared memory within our task framework, and enable users to implement data parallelism for a task on a distributed platform easily.

Besides data parallelism, task parallelism is another important path to develop parallel applications. To support task parallelism, OpenMP3.0 introduces the "#pragma task" directive to allow the user to identify a code block as a task that can run concurrently with the main program. OpenMP4.0 [35] introduces the "#pragma target" clause to support offloading code to accelerators. This ability relies on compiler support, such as Intel OpenMP for Intel Xeon Phi [36]. Some low level thread libraries can also be used to implement task parallelism. Linux system native thread library Pthreads is the most widely used. Many programming languages and methods make use of Pthreads library implement parallel execution. In our work we also use the Pthreads library to create parallel threads as the basic execution unit. Besides Pthreads other low level thread libraries like Intel Thread Building Block (TBB) [37], Qthread [38] are also used on different platforms. Using these low level thread libraries to implement parallel applications requires lots of effort by users. But using our framework, a user only needs to specify the number of threads when creating a task instance and the framework runtime system will spawn and launch the required threads for each task correctly and provide helper functions to do synchronization. So the user can focus on the application logic with little concern about thread creation and management.

The PGAS model introduces a distributed shared memory space to ease programming effort with a shared memory model and one-sided communications. It lacks the ability to dynamically spawn parallel activities, so Asynchronous PGAS (APGAS) [39] is proposed to improve the task parallel support. X10 [40] is a representative programming language of APGAS. By extending the Java language, X10 introduces *place* and *async* keywords for the user. *place* stands for an execution context that contains many ongoing activities, and usually, each node has a place where programs start running on that node. The runtime system will convert a function declared with *async* to a task (an activity) that will be scheduled to run asynchronously. A task can be spawned to run in any *place* and cannot migrate once started. Data are transmitted between different *places* with a one-sided communication pattern. Each task's input/output info must be provided by the user and the X10 runtime will use this info to build a data flow graph which is a Directed Acyclic Graph (DAG). Based on this graph, the task scheduler will schedule and run all tasks for better load balancing. In X10, a

task can only access the data of its resident *place*. To access remote data, one needs to first spawn a task with *async* on a remote *place* and then process the data access or transfer. So the programmer is primarily responsible for decomposing the application's data across the system and creating and organizing the asynchronous tasks accordingly. Using the same APGAS runtime infrastructure as X10, ClusterSs [41], a member of the StarSs program family [42], enables the user to select functions in a serial program, and run these functions asynchronously and concurrently based on a data flow driven model with the help of compiler and runtime system. OmpSs [43] is another programming method that aims to support task or function parallelism as well as high programmer productivity. OmpSs combines the ClusterSs's ability of asynchronous task creation and OpenMP's compiler directives features. Using directive statements, the compiler automatically generates programs with asynchronous tasks. Then tasks are scheduled and run on clusters the same as ClusterSs.

These programming methods are trying to help the user express task parallelism in an application more easily. The tasks created in these methods are usually very lightweight tasks, such as small functions or several lines of statements in a loop structure, so they are more like function parallelism. All tasks are run based on their data dependencies, so the compiler analysis and scheduler is critical to these systems. In our work, we are not focusing on fine-grained task scheduling and load balancing. Here we introduce a global shared memory mechanism based on our high level task design to enrich the task capability of expressing data parallelism. The task parallelism is explicitly expressed by user with the high level task components. We utilize these high level task abstractions to isolate the low level task implementation from applications' program structure for improved potability. Also these programming methods usually relies on their specific compiler's support which make them hard to change to fit various systems and platforms. In our work, programs are composed and compiled with standard C++ language and compiler, and linked with our framework runtime libraries to run. So it is easier for us to make changes to adapt to the underlying platforms, such as adding new features due to the improvement of underlying libraries or support new hardware like GPUs.

Besides parallel programming with traditional CPUs, programming with accelerators, especially GPUs, attracts a lot of research interest. CUDA and OpenCL are two primary programming approaches for GPUs. But producing high performed CUDA or OpenCL programs is not easy and usually require the users to be familiar with GPU hardware features. To ease programming difficulty, new programming language features and compilers are introduced to help user generate GPU programs automatically. OpenAcc [26], OpenMP and X10 to CUDA [44] allow users to insert compiler directives to let a compiler generate GPU programs automatically. But it is hard to guarantee

the quality of generated code [45]. Along with compiler approaches, library or algorithm template based approaches also reduce programming difficulty and can provide high performance, for example cuFFT and the Thrust library [46]. Similar to Thrust, SYCL [47] introduces specifications of a C++ template library and SYCL device compiler, which enable users to run C++ programs on OpenCL supported heterogeneous platforms. In our work, we adopt a hybrid programming approach to allow users to mix CUDA implemented computational kernels into specific task implementations to utilize GPUs for acceleration. Users are responsible for implementing the CUDA kernels, but the GPU execution environment is already setup by the framework runtime. So it is easy for users to make use of the available GPUs of a platform for kernel execution. Also, due to the hybrid programming pattern and task design, users can make use of existed CUDA programs or libraries like cuFFT and cuBLAS to realize their own kernels efficiently.

Features	Pthreads/ OpenMP	MPI	OpenSHMEM/ UPC	X10/ClusterSs/ OMPSs	PVTOL	UTC
Shared Memory	✓		✓	✓		✓
Distributed Memory		✓	✓	✓	✓	✓
One-sided Remote Memory Access		✓	✓	✓		✓
Dynamic Tasks Creation/Execution	✓	✓		✓	✓	✓
Fine-grained Task Scheduling				✓		
High-level Program Components					✓	✓
Heterogeneous Platform Support	✓	✓	✓	✓	✓	✓
Easily Extensible		✓	✓			✓

Figure 2.8: Features of Different Programming Tools and Methods

Generally, we introduce a unified tasks and conduits framework and implement a light weight runtime to help users develop parallel applications on heterogeneous platforms. Under this framework, the user expresses coarse-grained parallelism with multiple tasks. In each task, a user can further explore fine-grained data parallelism through multiple threads. We use the C++ language and implement our framework through a library-based approach, which is easier to extend to add new features, such as running tasks on GPUs. Compared to existed work, our approach combines different

features together, like shared memory and message passing, data parallelism and task parallel. Also we focus more on the portability and maintainability of parallel applications and aim to help user to develop well structured applications through this framework. A comparison of features of different programming tools and methods is show as Figure 2.8. No other approach is as comprehensive as UTC.

Chapter 3

Methodology and Design

This chapter introduces the methodology of designing the Unified Tasks and Conduits framework. To support the framework, we implement a thin runtime system prototype on heterogeneous cluster platforms. We will show the interfaces used to define and create task and conduit instances and how parallel threads are organized for each task. Also we provide a task based global shared memory for users to easily develop parallel applications on cluster platforms.

This chapter contains four subsections. 3.1 shows the overall design of the UTC framework and how parallel applications are organized based on this framework.

Section 3.2 details our implementation of the framework runtime system. We introduce the basic interfaces and functions provided for users to develop applications through this framework. We detail the tools and methods to realize parallel execution of tasks and data movement of conduits for cluster platforms.

In section 3.3 we describe the framework extensions to support creating and running GPU tasks. We provide an environment for users to invoke and run CUDA implemented GPU kernels to make use of available GPUs on the platform for acceleration.

Section 3.4 shows the design of task based global shared memory. We implement global shared data for users to run multi-threaded tasks on distributed platforms and share data sets through one-sided remote memory access. In this way, a single task is not limited to a single node and is able to scale to multiple nodes for better performance.

3.1 Unified Tasks and Conduits Framework

Developing high performance parallel applications on heterogeneous platforms is hard, and rapidly changing hardware features and platforms makes the situation even worse. The available parallel programming methods are usually low level. They focus more on data distribution and communications, aiming to generate high performance and scalable applications on specific platforms. The programs usually are closely related to the underlying platform. For an application composed of several workload parts, it is hard to use these parallel programming models to express the application at a higher level, which abstracts different workload parts as tasks and separates an application's high level structure from the lower level task implementations. Due to the low level features of traditional parallel programming methods, when the platform changes, lots of modifications to the application are required to target a new platform. Also, users may need to familiarize themselves with the hardware features of the platform to tune parallel applications. This makes it much harder for domain experts who lack hardware knowledge or are not familiar with novel processing devices to produce high quality parallel applications.

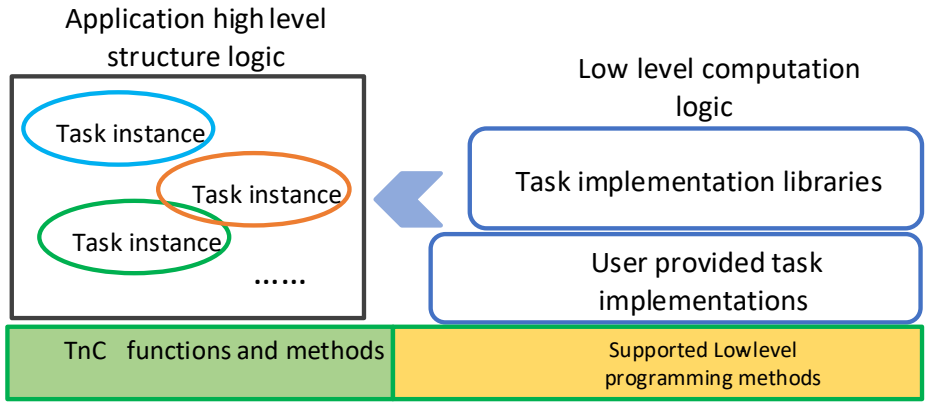


Figure 3.1: TNC Model Based Application

To reduce development effort and improve portability of applications, we need a programming method that can hide or separate low level platform features from the high level application implementations to ease the programming of parallel applications as well as maintain good performance. This is why the TNC model is introduced. In this model, tasks are the abstraction of computation workloads while conduits are the abstraction of data transfer between different tasks. A parallel application is comprised of tasks which are responsible for different computational work loads and can run in parallel, as shown in Figure 3.1. Through this model, we can decouple the

application structure from the hardware platform easily and conveniently.

Based on the TNC model and targeting heterogeneous parallel computing platforms, we propose the Unified Tasks and Conduits (UTC) framework. As shown in Figure 3.2, the UTC framework includes a series of interfaces/functions and a runtime library. Application developers define and implement necessary tasks and conduits to form parallel applications with the interfaces and functions, while the runtime system helps to set up the parallel execution and realize data movement on target platforms.

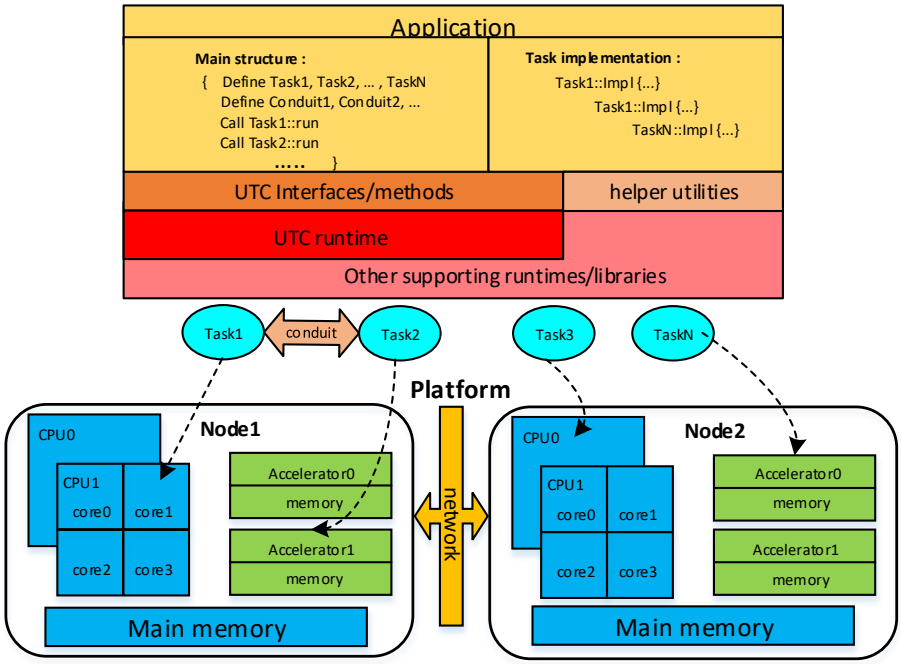


Figure 3.2: UTC Framework Overview

Using this framework, application development includes two levels. At the high level, the user mainly focuses on creating the necessary task/conduit components to construct the application, while at the low level, the user can implement each task through various programming methods based on the target platforms. Different task implementations can be easily organized in a library for reuse. In this way, an application's main structure can be expressed at the task level, which is independent from the task implementation and execution on a specific platform. For different platforms, we may have different task implementations. When running or porting such an application to different platforms, the application main structure can remain unchanged and only adopt appropriate task implementations, reducing the effort of developing and maintaining parallel applications.

3.2 Framework Implementation for Cluster Platforms

As described above, the UTC framework defines a set of interfaces/methods and needs a runtime library to support developing and running parallel applications. In this section, we discuss the details of framework design and runtime system implementation.

3.2.1 Framework Runtime System Design

Supporting software for framework implementation

Our framework aims to facilitate developing parallel applications, so there are two basic works we need to implement in this framework: create the parallel execution environment and realize the data movement for parallel programs. Also, our target platforms are GPU based heterogeneous cluster platforms, so there are various tools and libraries we can use to build up our framework runtime system. The overall framework software stack is shown as Figure 3.3.

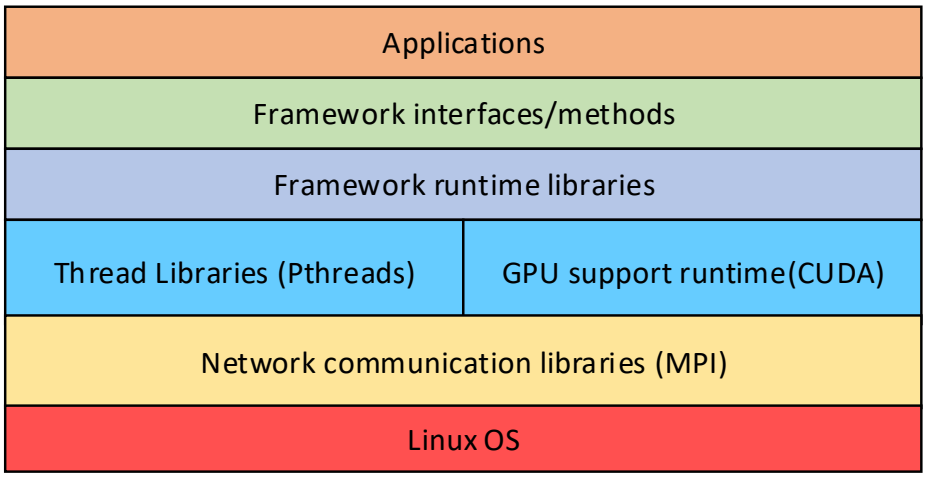


Figure 3.3: Framework Software Stack

In our design we use threads as the basic execution unit, so we need the support of thread libraries. Various thread libraries can be used [4, 38, 37]. Here we choose the Pthreads library as our underlying library runtime. Pthreads is the Linux system's native supplied standard thread library, so it is supported by a large number of Linux systems and platforms; it is a low level thread library and supports system related low level management such as CPU binding. The low level features also increase the difficulty of using Pthreads directly for application development. However, our framework runtime will take care of thread creation and management, providing simple methods

for a user to interact with multiple threads. Further, because we use system native threads as the basic execution unit, this gives users the flexibility to manipulate the threads for fine-grained low level operation. In addition to the thread library, we also rely on the CUDA runtime to provide GPU support in our framework. The CUDA programming language together with NVIDIA GPUs are the most popular programming method and accelerator in current high performance computers. There are rich CUDA applications and libraries for users to use for application development. We choose CUDA as the underlying GPU support software.

Our target platforms are distributed cluster platforms, so we need to be able to execute programs on multiple nodes and carry out data communication across the network. Several software packages can be used to accomplish these aims [48, 49]. In our framework implementation we use the MPI library. MPI is the most popular programming approach for distributed platforms and it fulfills our requirements. Through the MPI runtime we are able to launch processes on distributed nodes to initialize parallel execution on multiple nodes at the same time. More important, MPI defines and provides a rich set of communication methods for us to realize data communication. Based on MPI runtime, the complex network structure and fabric features are hidden from us, simplifying our framework runtime implementation.

Runtime design for multicore cluster platform

Based on the underlying software and libraries, we first design and implement a runtime system prototype for multicore cluster platforms. To provide high level abstract components (task/conduit) for implementing parallel applications in our framework, we leverage object oriented programming methods (C++). In object oriented programs, an object is the integration of data and specific operations on that data. This is well suited to the TNC model where tasks and conduits are treated as two basic objects. Combining C++ object design and parallel threads, the UTC runtime overall design is show in Figure 3.4.

In a task object, the data members contain source data and results data that will be accessed and used for computation, as well as some other necessary arguments. The member functions define and implement specific computational logic or algorithms. The conduit object works as the bridge between different task objects, containing internal buffers and providing methods for transferring data between task objects. The parallel execution flows are represented by multiple threads. Each task object is bound to one or several threads, and invokes threads to execute certain member functions.

Under this framework, when running an application, N processes (*UTC_process*) are started

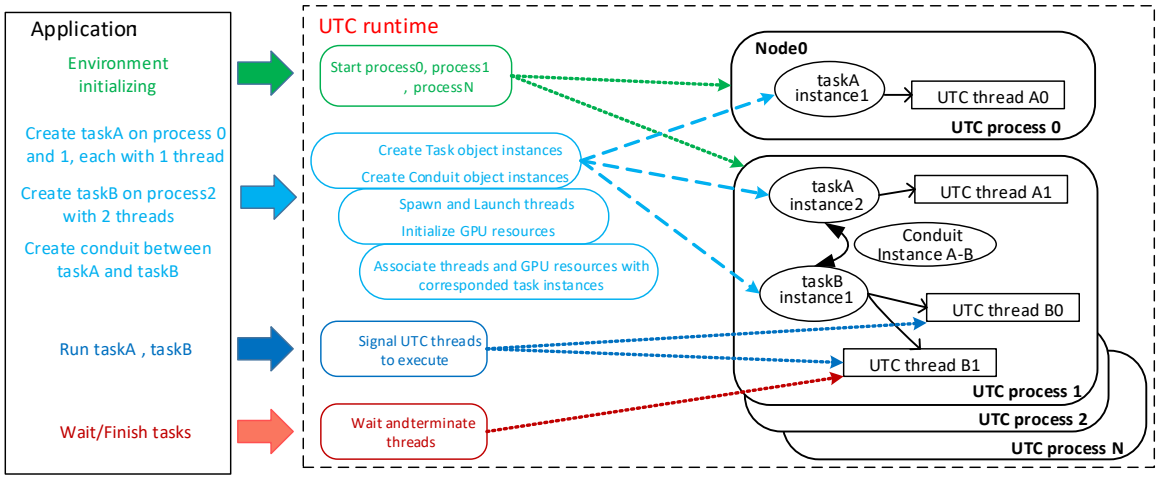


Figure 3.4: UTC runtime overall design

by the system and execute the application program in a SPMD pattern. Every process has a unique id (from 0 to N-1). In our design, we assign one process to a compute node, thus each process represents a unique node in the cluster. One *UTC_process* stands for one compute node.

1. When creating a task to run on a single node, a *task object* is instantiated in the *UTC_process* of that node. Also one or several threads can be launched in the same *UTC_process*. These threads serve this task and are bound with the *task object*, like "taskB" in Figure 3.4.

2. When creating a task to run on several nodes, in each *UTC_process* of these nodes, a *task object* is constructed and bound to the demanded number of threads, such as "taskA" in Figure 3.4.

So, in our framework, **a task is composed of one or multiple *task object* instances and a group of threads** which can perform computations in SPMD. Different tasks can perform parallel computations using the MPMD paradigm. Thus parallel applications can express both data parallelism and task parallelism easily and naturally in the UTC framework. In a program, the user needs to provide mapping information that indicates which nodes a certain task will be executed on. On each node, a thread can use any core for execution depending on the scheduler of the operating system. By default, multiple threads are not mapped to fixed processing cores. Our framework allows the user to bind threads to certain cores. With the help of this framework, we can define and create necessary task and conduit objects, and invoke predefined interfaces and methods to construct the main structure of an application. Then, in different tasks, we implement the necessary computational algorithms with supported runtime tools and libraries.

3.2.2 Basic Interfaces

As discussed above, task object and conduit object are two primary objects used in our runtime system. Along with them several methods are provided to perform certain operations. Some basic classes and interfaces defined in UTC are listed in Table 3.1. The user specifies a class which inherits the *UserTaskBase* class and realizes computational kernels under certain interfaces. This class can then be used as a class template (tepmlate T) to create *utc_task* objects. When a *utc_task* is constructed, a group of threads are also launched waiting for signals to perform the computational workload. With these classes and methods as well as other complementary utility functions, it is easy for a user to implement parallel application.

Table 3.1: Basic Classes and Methods

Class/Methods	**Description**
class UtcContext;	Construct and initialize UTC environment in program.
class<template T>Task;	Create a utc_task object with user supplied template T.
Task::init;	Signal task_threads to execute T::initImpl.
Task::run;	Signal task_threads to execute T::runImpl.
Task::wait;	Wait for task_threads' work complete.
Task::finish;	Terminate task_threads.
class Conduit;	Construct utc_conduit object between two tasks.
Conduit::read;	Fetch data from conduit object.
Conduit::write;	Put data to conduit object.
class UserTaskBase;	A interface class for user to implement specific workload.
UserTaskBase::initImpl;	Interface for user to implement initial work.
UserTaskBase::runImpl;	Interface for user to implement computation kernel work.

3.2.3 Task Implementation

Task creation

In the design of the UTC framework runtime, we start one process (*UTC_process*) on each compute node and then generate groups of threads in each process to run. To start up multiple *UTC_processes* executing both locally and remotely, we make use of an MPI library in our framework runtime. Applications are initialized through the mpirun command which launches a number of

processes on multiple nodes. Each *UTC_process* has a unique ID number which equals the MPI rank value of this process. After the *UTC_processes* are started, the user can simply define and create a task with:

Task<*USER_TASK*> task_instance(*proc_list*, ...);

There are two important required arguments:

1. *USER_TASK*: This is a user defined class template, inside of which the user's specific computational workloads are implemented. This class must implement certain interfaces, such as *initImpl/runImpl*, which are defined in the *UserTaskBase* interface class. Inside these interfaces, the user can realize the various computational algorithms. When the runtime instantiates a *task object* in a *UTC_process*, an object of this template class (*user task object*) is also created and cached in *task object*.

2. *proc_list*: This is a vector of *UTC_process* IDs. It indicates where and how many threads will be launched for this task.

After the task is created, all necessary threads are launched and suspended, waiting for commands from the application to perform a specific execution. There are four basic methods used to send commands to threads:

- *Task::init*: Signal threads to execute *initImpl*;
- *Task::run*: Signal threads to execute *runImpl*;
- *Task::wait*: Wait for threads' ongoing work to complete;
- *Task::finish*: Terminate associated threads;

The user invokes these methods to ask all threads bound to the calling *task object* to perform the required actions. *init/wait/finish* are synchronized methods. They will wait for the threads to finish a certain job. The *run* method is asynchronous and returns as soon as the command is submitted. So a user can invoke different tasks to run successively, without waiting for earlier tasks' completion.

CPU affinity for threads

On a multicore CPU platform, a process or thread can run with any available core and usually the OS may schedule and migrate a process/thread from one core to another. On a Non-uniform memory access (NUMA) system the migration may even happens between multiple CPUs.

By setting the affinity of each process/thread, it will always use the same core or a group of cores for execution and will not be scheduled and migrated to other cores. There is some research about CPU affinity setting and impact [50, 51]. Because we use Pthreads as our underlying threads library, and it allows us to set CPU affinity for threads. In our task creation procedure we also provide several basic configure options to set task threads' affinity attribute. We provide the following options:

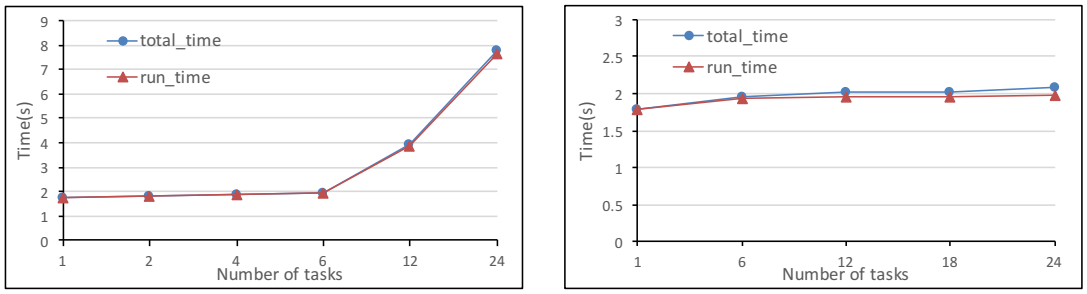
- **bind-to-all**: No affinity setting and thread may use any cores of any CPUs on a node;
- **bind-to-cpu**: A thread can use cores of same CPU;
- **bind-to-core**: A thread always runs on the same core;
- **bind-to-ht**: A thread runs with same hyperthread of a core;

When a user defines and creates a task, the affinity options can be set through configuration arguments and the runtime system will perform the affinity setting for task threads. The effect of different affinity settings depends on the programs' characteristics and no one approach is always the best choice. By default, bind-to-all is set for each task.

Task creation test

Based on our runtime implementation, we use a micro-benchmark to test task creation performance. In this test, we create a number of tasks; each task has one thread and completes the same amount of computation. We record *total_time* which includes the time to create all threads of all tasks and the time for computation. The *run_time* only counts the average value of one task thread's computing time. We run the test on a small cluster with 4 nodes where each node has a six core CPU. The test results are shown in Figure 3.5.

For the single node test, we run all tasks on one compute node. Because one task uses one thread and the node has a six core CPU, from one task to six tasks the measured time stays about the same. After six, the time increases as there are no more idle cores for concurrent thread execution. In the multi-node test, we have at most 6 tasks active on one node, and 24 tasks run on 4 nodes. All these tasks are able to run concurrently. From the results we see that *total_time* and *run_time* are very close which means the overhead of task thread creation is very small. When more tasks are launched, the overhead also increases. This is because the time for managing and synchronizing more threads increases. If the computational kernel is more time consuming, the overhead should be negligible.

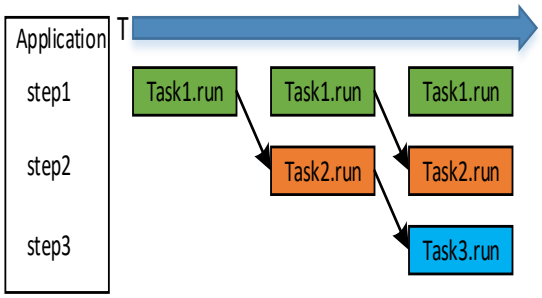


(a) (b)

Figure 3.5: task creation test: (a) single node;(b) multiple nodes (6 tasks per node).

Task pipeline execution

Through UTC, a user can define and create several tasks in an application. These tasks run independently or cooperatively in parallel. A complex application may includes several parts or sub-problems. Each part represents a UTC task and can be realized as a parallel program, using groups of threads running on the cluster. In each UTC task, the concurrent threads are tightly coupled to implement parallel algorithm logic. Between different UTC tasks, they are loosely coupled. They may collaborate through exchanging data using a conduit, or just run independently. A special case occurs when an application includes several consecutive parts. Then it is easy to create multiple tasks and implement a pipeline running pattern, as shown as Figure 3.6.



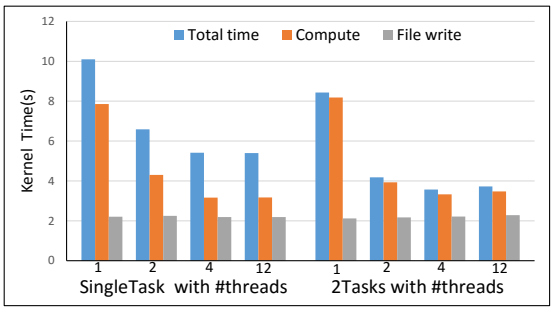


Figure 3.6: Task Pipeline Running

Figure 3.7: Pipeline Test

Figure 3.7 shows a pipeline execution test. In this test we define an application that has two steps of work: do some computational work and write result data to a file. We implement the application in two approaches. First, we use one task to compute and save data to files successively. Second, we create two tasks: one does the computation and then writes data to a conduit; the other

one reads data from the conduit and saves it to files. From the results we see that for the single task version, the total time is the same as the summation of data computing and file writing times. But with pipelining two tasks, the total time is only a little more than the computation time, which indicates that the file write procedure is hidden by computation.

3.2.4 Conduit Implementation

In UTC, we use conduits for communication between different tasks. A *Conduit* class is provided for users to define a *utc_coduit* object to connect two tasks. Then the user can invoke the read/write methods of the *utc_coduit* object to exchange data. The UTC runtime system supports two kinds of basic conduits: intra-proc conduit and inter-proc conduit. When constructing a *utc_coduit* object, the runtime system will automatically decide which kind of conduit is going to be used.

Intra-proc conduit

An intra-proc conduit connects two tasks that run on the same node. Because on each node we create one *UTC_process*, all tasks in a program that run on the same node live in the same process, sharing the same address space. For this reason, we implement conduits using the shared memory mechanism. The intra-proc conduit is show in Figure 3.8.

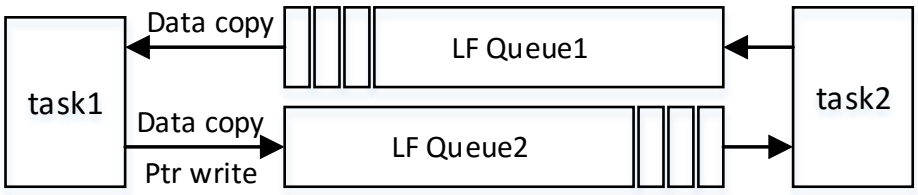


Figure 3.8: Intra-process Conduit

We set up two buffer queues in an intra-proc conduit, one for each direction of the data transfer. The two directions can advance at the same time. To reduce data transfer latency, we use lock-free methods to implement the shared buffer queues. Further, when moving data from one task to another, for small data sets we copy all the data to the shared buffer; for large data sets we only pass the source data address to the destination, avoiding memory copy operation.

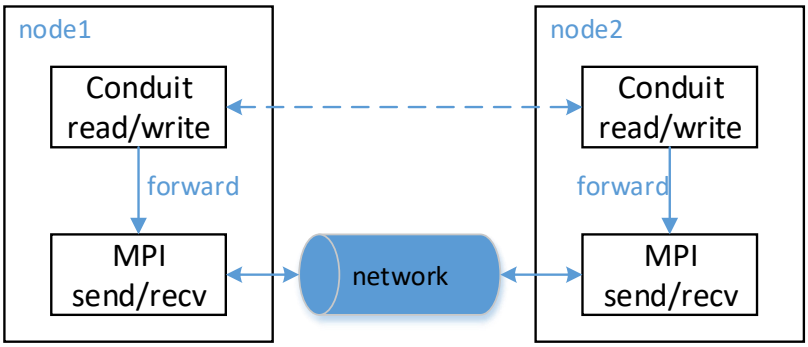


Figure 3.9: Inter-process Conduit

Inter-proc conduit

Inter-proc conduits connect two tasks that reside on different nodes. The data movement happens through the underlying network. As discussed before, we leverage the MPI communication libraries to achieve data transfer between nodes. When invoking conduit read/write operation to move data, we forward data set information such as address, size, and type to underlying MPI send/recv methods to realize inter-proc conduits, as shown in Figure 3.9. There are two issues to be considered when using MPI:

1. The task comprises a single or a group of threads. It should be safe to use MPI functions in the multithreaded situation. Most MPI implementations now support ”MPI_THREAD_MULTIPLE”, so it is safe to use MPI point-to-point communication in threads. However, there are restrictions and performance degradation compared to the pure process situations [52, 53].

2. All tasks on the same node have the same MPI process rank value, so they need to be differentiated when doing inter-proc communication with the same destination. We use MPI message tags to solve this problem. Every task has a unique task id, which we append to the tag, reserving the lower eight bits of the tag to store this id, when invoking MPI send/receive functions. In this way, different tasks are ensured to have different tags, and will correctly support MPI message matching operation.

Data movement between multi-nodes tasks

Intra-node conduits and inter-node conduits support data movement between two tasks that are either on the same node or different nodes. In both cases, we assume that each task is active on one node, and there is only one entry point to the conduit for each task. So the data transfer of the conduit happens between one pair of nodes. However, if the task spans multiple nodes, the conduit

between such tasks is more complicated.

When source task (S-task) and destination task (D-task) runs on multiple nodes, the source data may come from any S-task nodes and the destination may be any D-task nodes. Assuming S-task runs on N nodes, and D-task runs on M nodes, there are a total of N*M different possible transfer pairs. Usually, different applications may require different transfer schemes, so it is very hard to build a general node-pair scheme in a conduit which can adapt to any application's use case. Here, we provide two solutions for this problem, as shown in Figure 3.10.

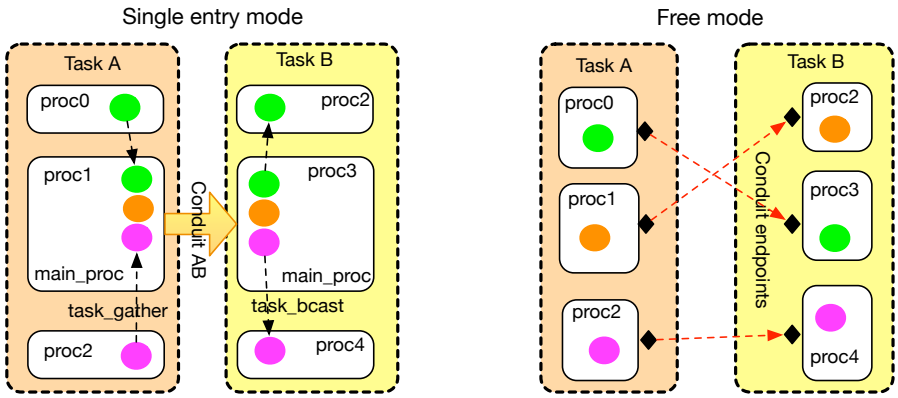


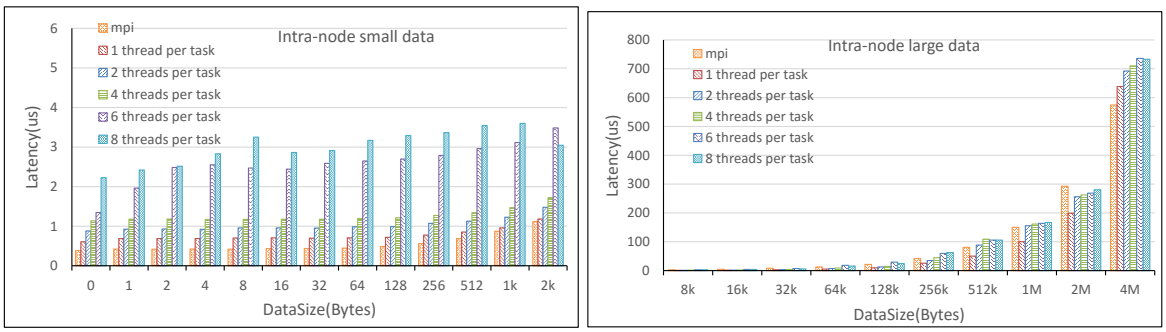
Figure 3.10: Data Movement Approach Between Multi-node Tasks

- **Single-entry mode**: In this case, for each task, we will select a process as the main process (*Main-proc*). Then the conduit is set up between these two *Main-procs*. Data are always transmitted between these two processes, which means a pair of designated nodes. Users need to aggregate data sets from different nodes to the *Main-proc* node in the S-task and send to the D-task. Then in the D-task, received data sets will be scattered to the right location. This is our default approach to choose the proper conduit when creating a conduit for two tasks.

- **Free mode**: In this case, an explicit two-sided point-to-point communication pattern is conducted. S-task initializes a data transfer to D-task with specific node ID and D-task also issue a receive call with correct source node ID. Data can transmit between any pair of nodes for two tasks, and different pairs of nodes can support the transfer concurrently. This process is the same as traditional MPI send/recv procedure, which makes data communication operations more complex and lacks the abstraction feature of conduits.

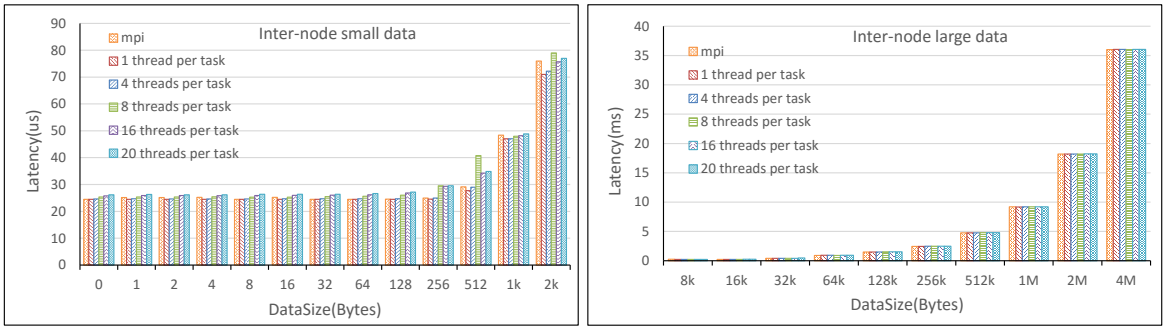
Conduit performance test

We developed a micro-benchmark to test the data transfer latency using conduits. In this test, we create two tasks to send and receive data in a ping-pong style. For different size data sets, we do the transfer a large number of times and calculate the average latency of moving data from one task to another. We also change the number of *task_threads* in each task instance to see how this affects the latency. For comparison, we do similar experiments using pure MPI send/recv with two processes. The results are shown in Figure 3.11 and Figure 3.12.



(a) (b)

Figure 3.11: Intra-node Conduit Latency



(a) (b)

Figure 3.12: Inter-node Conduit Latency

For the intra-node tests, MPI performs a little better than UTC when the data set is small. Both MPI and our implementation use the shared memory mechanism to realize data movement, thus the latency is much smaller than inter-node transfer, which depends primarily on network speed. For inter-node transfers, because the UTC runtime makes use of MPI to implement data communication

between nodes, the test results are almost the same as MPI. From the results we find that when increasing the number of threads in a task, the latency also increases. This is because if the task is a multithreaded parallel program, all threads run in the SPMD pattern. When doing conduit read/write, we only choose one thread to execute the operation, while other threads will wait and synchronize. These overheads increase with increasing numbers of threads. However, for large data set transfer or for inter-node transfer, where the time of moving data overwhelms these overheads, the effect should be negligible.

3.2.5 Code Sample

Based on the introduced interfaces and runtime libraries we are able to program parallel applications through the UTC framework. Figure 3.13 shows a sample basic program using our framework.

```
class DataInput: public UserTaskBase{
    initImpl(...){/* some initial work*/ }
    runImpl(){
        /* prepare the input dataset;
           write to destination through cdt; */
    }
    Conduit &cdt;
}
class DataOutput: public UserTaskBase{
    initImpl(...){/* some initial work*/ }
    runImpl(...){
        /* get dataset through cdt;
           output the data; */
    }
    Conduit &cdt;
}
class DataAnalysis: public UserTaskBase{
    initImpl(...){/* some initial work*/ }
    runImpl(){
        /* get dataset through cdt_in;
           analyze the dataset;
           write to destination through cdt_out */
    }
    Conduit &cdt_in;
    Conduit &cdt_out;
}
```

```
int main(){
    //initialize utc context
    UtcContex ctx(...);

    // creating tasks and conduits
    Task<DataInput>    dit(proclist, tasktype, ...);
    Task<DataOutput>   dot(proclist, tasktype, ...);
    Task<DataAnalysis> dat(proclist, tasktype, ...);
    Conduit   cdt1(&dit, &dat);
    Conduit   cdt2(&dat, &dot);

    //init tasks
    dit.init(cdt1, ...);
    dat.init(cdt1, cdt2, ...);
    dot.init(cdt2, ...);

    //run tasks
    dit.run(...);
    dat.run(...);
    dot.run(...)

    //wait and finish
    dit.wait();  dat.wait(); dot.wait();
    dit.finish();  dat.finish(); dot.finish();
    ....
}
```

Figure 3.13: Program Sample Under UTC

As shown in the sample, in the left part we define and implement 3 different task templates. In each task class we complete some job under the required interface: *initImpl/runImpl*. Each task may also have conduit references that can be used for data transfer between task instances. The right part shows the main program. In the main program we use the three task class template to instantiate three task instances. When defining task instances, we can use arguments to tell the runtime how many threads are required by this task and other useful information. Besides tasks, we also create conduit instances and the arguments indicate which tasks this conduit connects. After the necessary tasks and conduits are defined, we can invoke the predefined methods to run these tasks. Finally we use the wait method to wait for the completion of each task's work and use the finish method to terminate the threads spawned for the related tasks.

The program structure is clear and concise. Low level computation logic and parallel algorithms for each computation kernel are implemented and wrapped in task implementation class templates. Main programs primarily handle the high level logic and have less relationship to the underlying hardware platforms compared with traditional parallel programs.

3.3 Runtime Extension for GPU Support

A major benefit of the UTC framework is that it is not limited to specific platforms; tasks can be either programs running on traditional CPUs, or kernels programmed with special languages or methods and run on accelerators. Here we target NVIDIA GPUs and make use of CUDA programming to extend our runtime implementation.

3.3.1 GPU Program Structure and Hybrid Programming

To enable GPU tasks, we need to integrate GPU programs into our task implementation. CUDA provides a runtime library as well as a variety of functions to implement and run programs on GPUs. The general GPU program structure is shown in Figure 3.14.

The program is divided into two parts: Host-part and Device-part. The Device-part comprises actual kernels which are compiled by dedicated compilers and converted to GPU supported ISAs. The Host-part runs on CPUs and plays the role of the controller: selecting and initializing the GPU, launching kernels on the GPU and copying necessary data to and from the GPU. Our current task implementation can be easily adopted to play the role of Host-part for a GPU task. In our approach, when users implement a GPU task they use a UTC+CUDA hybrid programming

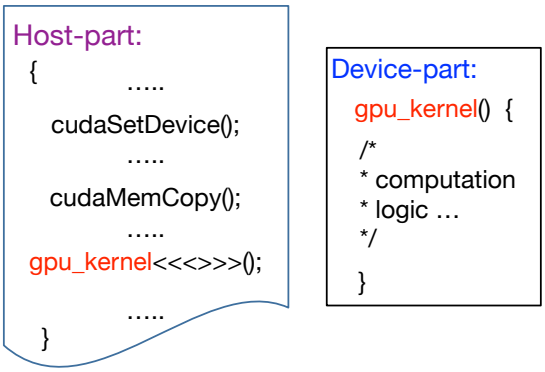


Figure 3.14: CUDA Program Structure

approach to write their code. The code that runs on the CPU behaves the same as a normal CPU task except that the GPU kernels which are programmed with CUDA will be invoked for running on GPUs. The interface of creating a GPU task is the same as before, but we add a task configuration argument to identify the task type: Task<*USER_TASK*> task_instance(*proc_list*, **"cpu_task"**, ...);

Task<*USER_TASK*> task_instance(*proc_list*, **"gpu_task"**, ...);

As soon as a GPU task is instantiated, the runtime will select and initialize the GPU execution environment. A GPU task can contain multiple host threads, in which case each host thread will be bound to one available GPU of the platform. By default, on one node we will select GPUs in a round robin fashion if there are multiple GPUs on the node. Users only invoke their GPU kernels in the task codes and there is no need for the user to consider the GPU management process.

GPU task test

Here we show some tests for creating and running GPU tasks. We developed four applications: Matrix Multiply(MM), 2D Heat Conduction(2Dheat), Heat Image Generation(Heatimage) and K-means clustering(Kmeans) with both CPU task and GPU task implementations. The CPU task versions are tested on a four nodes cluster which each node has a 6 core CPU with hyper-threading. The GPU task versions are tested on a server equipped with a NVIDIA Tesla C2070 GPU. The applications' speedup compared to sequential programs is shown in Figure 3.15. The CPU task uses 48 threads for parallel processing on 4 nodes. From the test results, we can clearly see the benefit of using GPUs for acceleration. For MM, one single GPU's performance is much better than 48 CPU threads; For others, one GPU's performance is comparable or better than 48 CPU threads.

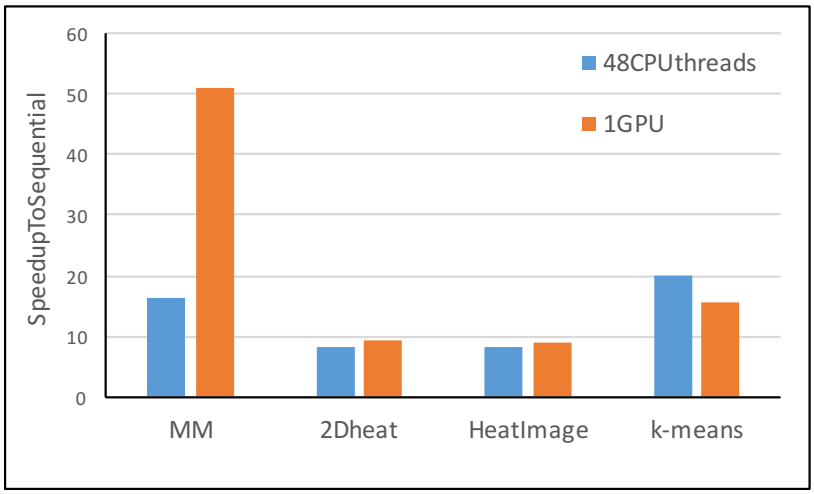


Figure 3.15: Comparison of GPU/CPU Task Execution

3.3.2 Concurrent Execution on GPU

In early generations of NVIDIA GPUs, only one kernel program can be executed at a time. Multiple kernels are queued and executed by a GPU sequentially. However, from the Fermi GPU, NVIDIA supports running multiple kernels at the same time on one GPU if the hardware resources are available. With the enhancement of single GPU's computing resources (the NVIDIA GP100 GPU has 3840 CUDA cores), it is common that one kernel cannot exhaust all the resources at a given time. So running multiple kernels at the same time can make use of GPU resources more efficiently.

In CUDA programming, when initializing a GPU, a CUDA context is created. The following GPU related operations are associated with this context. There may be several CUDA contexts for the same GPU at the same time, but only one context is active at one time. A GPU will switch between these contexts frequently to process each context's operations. In one CUDA context, a user can have multiple CUDA streams. A CUDA stream comprises a series of GPU operations, such as memory copy and kernel execution, and they are executed one by one in a stream. Different streams of the same context can be processed in parallel. So different GPU operations of different CUDA streams can be executed simultaneously, as shown in Figure 3.16.

In our GPU task design, each task runs with single or multiple threads asynchronously and simultaneously. Task threads serve as the host control part for a GPU program and are associated with a GPU device for later GPU kernel execution. When different task threads are bound to different GPUs, GPU kernels of different task threads run in parallel. Furthermore, when they are bound to the

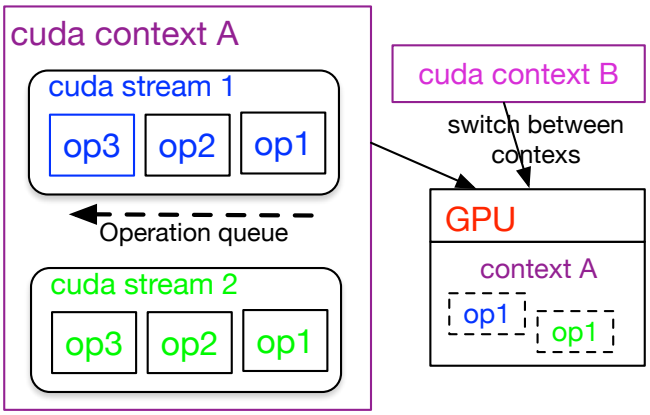


Figure 3.16: GPU Stream for Parallel Execution

same GPU, we can leverage CUDA streams to explore concurrent kernel execution. To make use of CUDA streams, for each GPU task we define a class *GpuContext* to record useful information related to a GPU device as shown below. When a GPU task is created, a `GpuContext` object is also setup.

```
class GpuContext{
    ...
    unsigned int    gpuId;
    cuContext_t     cudaCtx;
    CudaStream_t    streamId;
    ...
    }
```

In this object, GPU ID indicates which GPU in the system a host thread is bound to. Multiple host threads may bind to the same GPU. The CUDA context handle (*cudaCtx*) tells the CUDA runtime environment the proper CUDA context to activate. The stream handle refers to the unique CUDA stream for the related GPU task thread. In the GPU task implementation, a user will use this stream as the argument for GPU operations. To target a single GPU, there are two mapping methods between a *GpuContxt* and a CUDA context:

1. One to one mapping: each *GpuContxt* refers to a unique CUDA context. In this way, each GPU task will use a different CUDA context and they share a single GPU by context switching.

2. One to many mapping: multiple *GpuContxt* refers to the same CUDA context on a single GPU. However, different *GpuContxt* still use different CUDA streams. So different GPU tasks can

execute on a single GPU concurrently with the help of multiple streams. This is the default configuration.

Based on task asynchronous execution and multiple streams binding, we can easily run GPU kernels in parallel on either a single GPU or multiple GPUs.

Test of concurrent task on a single GPU

Here we use a Nbody simulation application as the test program and run on a NVIDIA Tesla K40m GPU. In Nbody simulation, when the total number of bodies is small, one GPU kernel can not consume all the computing resources of one GPU and multiple kernels can execute simultaneously.

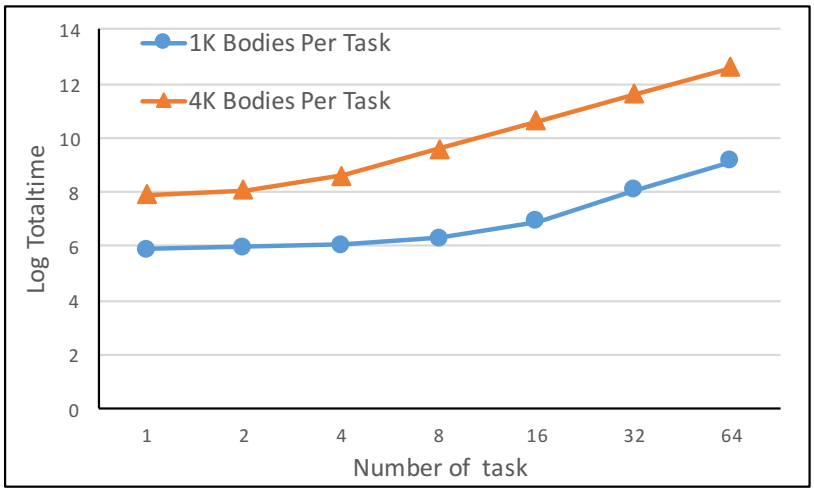


Figure 3.17: Nbody Concurrent GPU Kernel Execution

Figure 3.17 shows the program run time results, noticing that the time is plotted on a logarithmic scale. We create different numbers of GPU tasks and run, giving each task the same number of bodies. When the number of GPU tasks doubles, the total computation workload of the application also doubles. As 4K bodies are 4 times larger than 1K bodies, the GPU kernel under 4K bodies consume more GPU resources and fewer GPU tasks are able to execute in parallel. Our test results reflect this relationship: for 1K Bodies, the run time starts increasing noticeably after 8 GPU tasks; While for 4K bodies the run time starts growing after about 2 tasks.

3.3.3 Uniform Data Allocation for Data Transfer

A GPU is used as an accelerator (device) and usually connected to the system (host) by PCI-e. It has its own memory for accessing data while running programs. So any GPU application has two primary phases shown in Figure 3.18: (1) transfer required input data and results between system main memory (host memory) and GPU memory (device memory); (2) execute GPU kernels with GPU processing units. Targeting NVIDIA GPUs and with the help of the CUDA runtime system, users can allocate three different types of memory in a system for communication between host and device through PCI-e: pageable memory, pinned memory and unified memory.

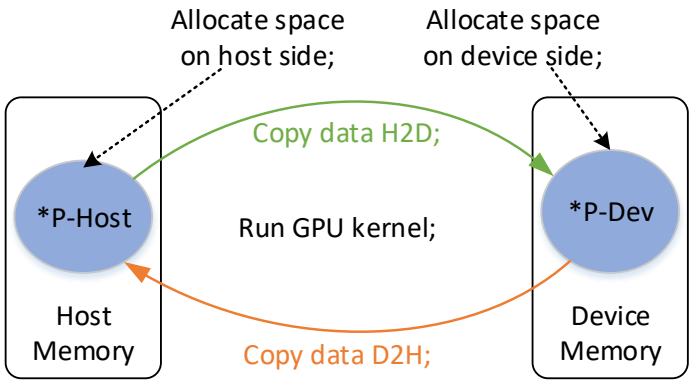


Figure 3.18: GPU Program Procedure

1. Pageable memory allows users to allocate memory blocks in virtual address space, and the allocated memory size can be larger than the available physical RAM size. The OS manages user allocated memory through the memory page mapping mechanism: when a program accesses a virtual address that is not mapped to physical RAM, a memory page will be mapped and transfer the data to physical RAM (page-in); when a memory page on physical RAM is not accessed for a while, it will be unmapped from physical RAM and the data cached on disk (page-out). So pageable memory is also called non-locked memory, which means it may not constantly reside in physical RAM. The drawback of using pageable memory for host/device transfer is that the communication procedure needs CPU involvement and possible page-in/page-out operations introduce extra overhead. So the bandwidth of the PCI-e bus that connects GPU to host is not fully exploited.

2. Pinned memory which can be allocated by CUDA utilities *cudaMallocHost* or *cudaHostAlloc* is locked memory, which means it cannot be swapped out (page-out) from physical RAM once it is allocated. Because it resides in physical RAM permanently, data transfer between host and device can be carried out by DMA (Direct Memory Access) operations. This can enhance the

bandwidth utilization of PCI-e and speed up the data transfer process. However, physical RAM is a limited resource. Allocating too much pinned memory could consume too much available physical memory and affect other programs or the entire system performance negatively. Figure 3.19 shows host/device and device/host memory copy bandwidth using pageable memory and pinned memory respectively. We can see that pinned memory has a better bandwidth performance than pageable memory.

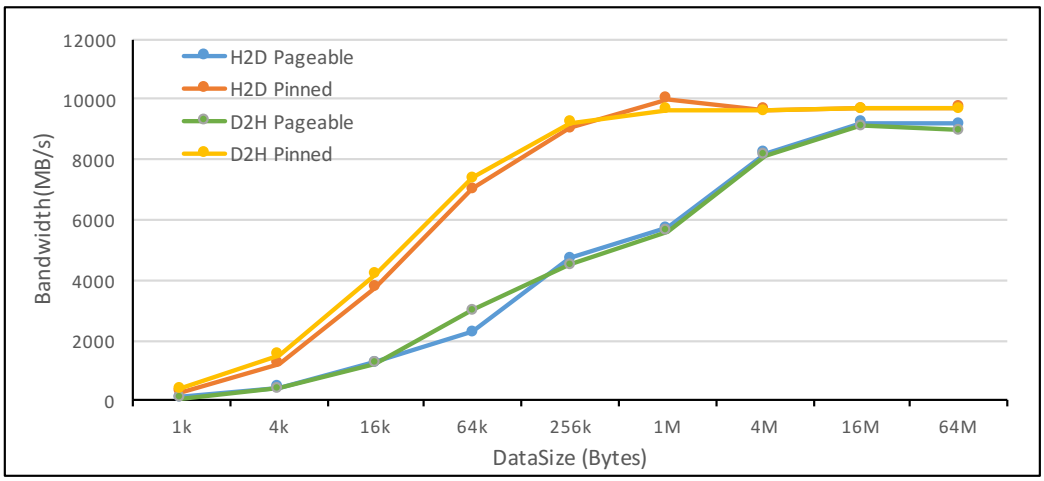


Figure 3.19: GPU Host/device Bandwidth with Pageable/Pinned Memory

3. In addition to pageable and pinned memory, starting from CUDA 6.0, unified memory has been introduced. Unified memory is memory blocks allocated using the *cudaMallocManaged* function. With the previous two types of memory, host and device have different address pointers and explicit data transfer operations are needed. In contrast, the unified memory system creates and manages a pool of memory blocks that are shared between CPU and GPU. There is only one address pointer exposed to the user and both programs on host and device can use this pointer to access data. Data transfer between host and device happens implicitly in the underlying CUDA runtime. Unified memory frees a user from tedious host/device memory creation and management, easing GPU programming. But unified memory management is not always as efficient as explicit memory transfer assigned by a user manually.

As described above, different kinds of memory can be used in a GPU program, and no single approach is the best choice all the time. Pinned memory provides better bandwidth usage, but we should avoid using it too much. Unified memory eases programming, but may not deliver the best performance. Also, using different types of memory requires different operations. The allocation and

management of more and more different types of memory blocks is error prone and burdensome. Therefore, to facilitate GPU task implementation and enable a user to explore different types of memory freely and conveniently, we provide a simple and uniform interface to be used in GPU task implementations. In general, we defined following classes:

```
enum class MemType_t{
    pageable,
    pinned,
    unified
    }
template <typename T>
class GpuMem{
    ...
    MemType_t   m_memType;
    T           *m_hostPtr;
    T           *m_devPtr;
    ...
    }
int   GpuMem::getH();
int   GpuMem::getD();
int   GpuMem::putH();
int   GpuMem::putD();
void  GpuMem::Sync();
...
```

When we need to allocate memory for a data set that is used on a GPU, we can create a *GpuMem* object with the desired memory type (pageable, pinned or unified). The object contains two pointers. When the memory type is pageable or pinned, the two corresponding pointers have different values and refer to host memory and device memory separately. When memory type is unified, these two pointers have the same value. Along with the object, we have different methods to get proper pointers and complete host/device data synchronization. Through these helper classes and methods, we can create a GPU task with appropriate memory type arguments to make use of different memory schemes. Once the GPU task program is implemented, users are able to try different types of memory for execution through an argument in task creation, without changes to the existing task implementation.

Task<*USER_TASK*> task_instance(...,***gpu_task, MemType_t::pinned***, ...);

3.4 Task Based Global Shared Memory on Cluster Platform

As discussed in 3.2.1, a UTC task can be a multi-threaded program that runs on multiple nodes. When a task is instantiated, a number of user task objects are created and a group of threads are launched in the system. Each node in the task's span holds one user task object and a sub-group of threads. We call all threads created for a task *global_thread_group* and call a sub-group of threads on the same node *local_thread_group* as shown in Figure3.20. From the perspective of a single thread, *local_thread_group* and *global_thread_group* there are three different data scopes in a task:

- private scope: data accessed by single thread and not shared
- local shared scope: data accessed and shared by *local_thread_group*
- global shared scope: data accessed and shared by *global_thread_group*

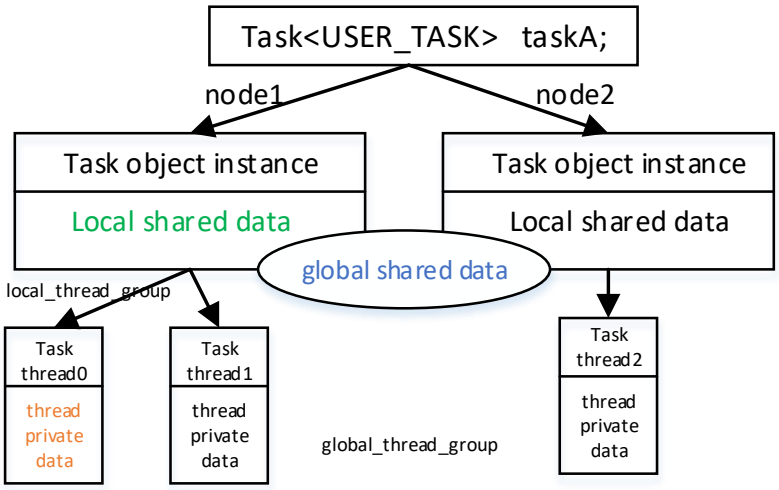


Figure 3.20: Data Scopes in a Task

Local shared data can be easily implemented in our design. Because *local_thread_group* is associated with the same task object on one node, all the data sets defined in this task object will be shared by local threads. We implement global shared data based on one-sided RMA operation. In this way we provide a hybrid local and global shared memory mechanism to enable users to explore data parallelism for task implementation on both single node and multiple node platforms easily.

The key point to implementing global shared data is to leverage one-sided RMA operation to complete inter-node communication. In this work we use MPI-3 to achieve this. As discussed

above, we have one process running on one node and different tasks assigned to the same node will reside in the same process and use different threads for parallel processing. MPI treats a process as the basic parallel processing unit. Using threads in MPI processes requires coordinating message passing procedure carefully to avoid conflicts and errors. We implement a global shared data object to hide the error prone procedure of managing MPI processes, windows and communicators together with threads context, providing concise methods to use RMA functions in task threads.

3.4.1 Remote Memory Access Through MPI One-sided Communication

One-sided communication in MPI-3

There are several network communication tools or libraries we can use for RMA, such as GASNet [30] which is a low-level network data transfer library used by many PGAS programming runtime systems or OpenSHMEM [23] which provides high-level one-sided communication methods for data transfer. Here we choose MPI as our main communication substrate to realize the global shared data feature. This is mainly because MPI is the de-facto communication standard and most popular parallel programming method for distributed platforms. It is supported by various network hardware and can deliver good inter-node communication performance. In addition, we already make use of MPI to implement the conduit behavior, and using the same underlying tool can simplify our runtime implementation and avoid possible inconsistency of mixing different communication libraries.

MPI standard 2 (MPI-2) started supporting one-sided communication patterns and in MPI standard 3 (MPI-3) this ability is further improved. In MPI one-sided communication, the critical concept is *MPI_Window*. By creating *MPI_Window* within an associated communicator, each MPI process in this communicator can expose a block of memory region to all other processes of the communicator for RMA. Based on the communication synchronization semantics, MPI supports two different RMA modes, Passive and Active[54]:

- Active mode: Using Post/Start/Wait/Complete (PSWC) to start and end RMA period. Under this mode, process A calls Post to expose its memory region to process B and B calls Start as a response to the beginning of this RMA period; B can then access A's exposed memory region. After a serious of RMA operations, B calls Complete to end this RMA period and A will call Wait to finish. By conforming to this complete PSWC process, MPI can ensure the data integrity and correctness of RMA operation.

- Passive mode: Using MPI window Lock/Unlock or LockAll/UnlockAll to start and end a RMA period. Under this mode, one process (caller) can call Lock or LockAll to start a RMA period for target processes within a specified communicator. After the RMA operations, the caller process will invoke Unlock or UnlockAll to close this RMA period. During the period, users can use flush functions to ensure the completion of RMA operations.

Compared to Active mode, Passive mode does not require the target process to participate explicitly. Therefore, Passive mode is closer to the shared memory semantic concept and we use the Passive MPI one-sided communication pattern in this work to realize the global shared memory feature.

Task based MPI window and shared space creation

When creating a task instance, there is a node list indicating which node and how many threads are required for this task. Because we have a one to one mapping between nodes and MPI processes, the node list just shows the MPI processes a task spans on. Different tasks on the same node will shared the same MPI process. When two threads of different tasks invoke MPI runtime for communication at the same time, message and operation mismatching may happen. To solve this problem, we make use of an MPI communicator to separate different tasks' communication contexts even if they are in the same MPI process. When initializing a task, we also create a MPI communicator, which includes the processes the task runs on, and cache this communicator handle in each task object (see Figure 3.21). This communicator handle will later be referenced to create an MPI window and carry out RMA operations. With the communicator, we can differentiate communication requests of different tasks in the underlying MPI runtime system correctly and enable them to progress concurrently.

Figure 3.22 shows the overall procedure flow of creating global shared memory for a task. Once the communicator is created, the next critical step is to create an MPI window and allocate memory space for each task. Similar with MPI communicator, different tasks have their own MPI window objects and memory blocks associated with corresponding MPI communicators. In order to support dynamically creating global shared data objects we adopt a memory reservation and sub-space reallocation approach: along with an MPI window, we allocate a block of memory by calling the OS's default memory allocation function. This memory region is bound with a task's MPI window and exposed as global shared memory space. In the task implementation program, whenever it defines a global shared data object, we will reallocate a sub memory block from the

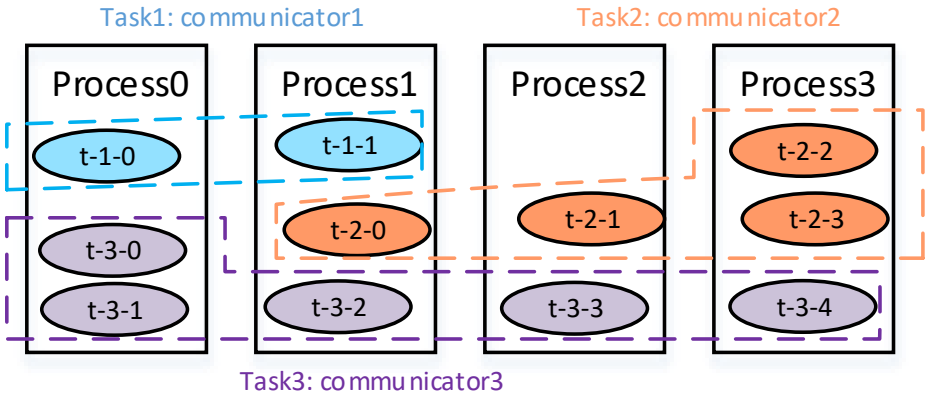


Figure 3.21: Task Based MPI Communicator

previously reserved memory region. This sub-space reallocation is accomplished through a set of non-OS managed memory allocation functions.

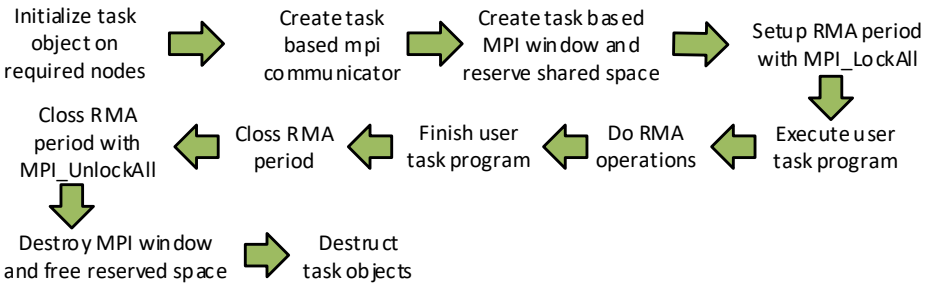


Figure 3.22: Shared Memory Setup/Finish Flow in a Task

When the task based MPI window and global shared memory spaces are ready, we can setup the RMA environment for a task. Based on Passive mode, we will invoke *MPI_LockAll* to expose the memory region of one process to all other processes of a task to start the RMA period. Then users can do RMA operations safely and correctly during the execution of a task. After a task finishes execution, we will use *MPI_UnlockAll* to end the RMA period as well as freeing and recycling all the resources we have created.

3.4.2 Global Shared Data Object

We defined a simple interface to create a global shared data object as needed in task implementations:

```
template <typename T>
```

```
class GlobalScopedData{
    int          size;
    size_t       offset;
    T            *data_ptr;
    MPI_Comm     *taskComm;
    MPI_win      *taskWin;
    ...
    }
```

taskComm and *taskWin* refer to the MPI communicator and window we create for a task in a MPI process and will be used in RMA operations. The *data_ptr* points to the sub-memory block allocated for a shared data object. The offset indicates the distance between the sub-memory block's starting address and base address of global shared space. Several methods are also implemented for accessing data stored in this shared data object:

- load/store: accessing data stored locally on one node.
- rload/rstore: accessing data stored on a remote node.
- barrier/quiet: synchronizing or waiting for completion of data movement

When using rload/rstore methods, the remote node id and accessed data size are needed. In our implementation, rload is a blocking method, which means when the function call returns, the data sets are moved from the remote node to the local node and are ready for use; rstore is a non-blocking method in which the function call return only means the data sets are ready to be delivered to the remote node, but whether the data movement is finished is unknown. To ensure the completion of rstore, the user needs to call synchronization methods explicitly. Using this shared data object and related methods, one can share data sets among multiple node easily and conveniently.

A task may have multiple threads in the same MPI process and these threads share the same shared data object. To avoid the conflict of concurrent access we also make use of a mutex to guard a shared object. Then one thread of a task at one time only can utilize a shared data object. Also, because threads in a process share the same data space, a user only needs to ask one task thread to do the RMA operation for a shared data object and the result can be seen by other threads.

In addition to sharing data sets within a task between different nodes through one-sided RMA operation, we also provide a set of utilities for collective communication behavior within a task. Based on the task-based MPI communicator we have built up, realizing task scoped collective communication is straight forward. We wrap up the collective communication information and

forward to the underlying MPI runtime to complete these collective communications. Currently we provided some frequently used collective operations like:

```
int  TaskBcastBy<type,..>(dataptr, count, ...);
int  TaskGatherBy<type,..>(dataptr, count, ...);
int  TaskScatterBy<type,..>(dataptr, count, ...);
int  TaskReduceSumBy<type,..>(dataptr, count, ...);
```

Shared memory for GPU support

Based on our framework design, we also support creating and running GPU tasks. For a GPU task, each task thread is associated with a CUDA stream on a selected GPU, playing a controller role and executing the host code of a GPU program. Since GPU memory is separated from system main memory it is impossible to use a RMA method to access remote GPU's memory directly. Depending on the topology of multiple GPUs in the system, there are two situations we need to consider:

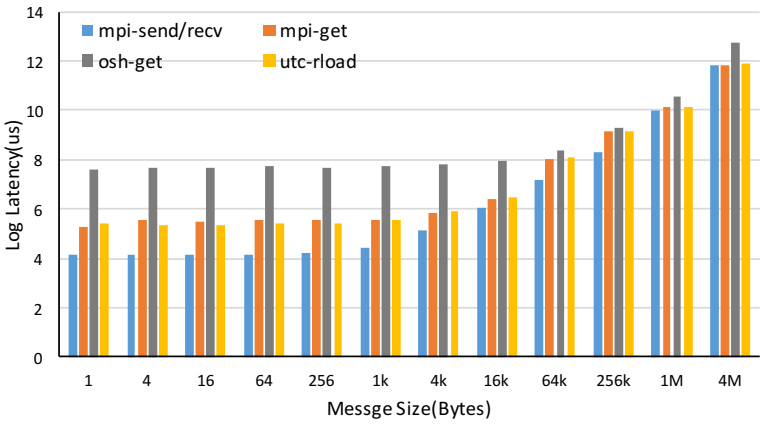
1. Benefiting from recent GPU hardware and software improvement, especially CUDA unified virtual address (UVA) and unified memory [55] feature, we can use a single memory address pointer for data accessed by different GPUs as well as CPUs. The data synchronization and consistency can be properly maintained by the user and the underlying CUDA runtime. As we use multiple threads to manage and control multiple GPUs on a single node, we can easily share the same address pointer between threads and accomplish data sharing between GPUs.

2. Programs running on one node usually cannot access GPU memory of the remote node directly, so we have to use system main memory to create an intermediate buffer to complete data sharing between distributed GPUs. As we already implement global shared data object in system main memory for inter-node data sharing, a user can use the shared data object as the intermediate buffer to transfer data explicitly between distributed GPUs. However, RMA operation only get/put data sets from/to remote node's main memory. To make the data available on GPUs, local side or remote side still needs another main-memory/device-memory movement operation.

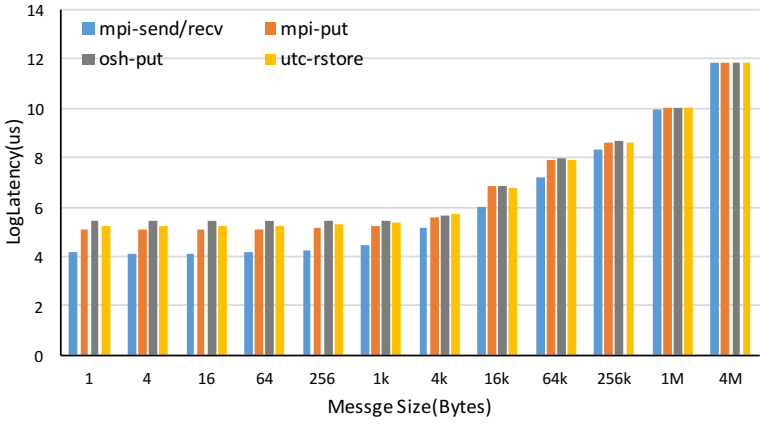
3.4.3 Global Shared Data Performance

We first used a micro benchmark to test the one-sided communication latency of the global shared data object. There are two basic RMA operations: reading data from remote memory(get/load)

and writing data to remote memory(put/store). As discussed earlier, remote write operation is an asynchronous non-blocking method, so we will do an explicit synchronize after we do remote write to ensure the completion of data movement. We also run similar tests with MPI send/recv, and MPI native put/get. OpenMPI includes an implementation of OpenSHMEM, so we also tested the latencies of using OSH put/get one-sided RMA. The results are shown in Figure 3.23. The Y axis is the latency in microseconds normalized with log_2.



(a) Remote Read



(b) Remote Write

Figure 3.23: Remote Memory Access Latency

We can see that explicit MPI two sided communications (send/recv) usually have the loweast latency results, especially for small size of message. This is mainly because with two sided send/recv, the receiver side can actively perform a quicker response compared to the one-sided pattern where there is no explicit receiver and depends on passive communication progress on the remote

node. The global shard data object has similar performance to using MPI get/put directly, which indicates that we did not add much overhead to make use of the underlying MPI functions. OSH get operation has the worse performance, though it uses the same underlying communication substrate as MPI.

In the UTC framework we introduced a hybrid shared memory mechanism based on our high-level task design to help users develop parallel applications on heterogeneous platforms. Leveraging the one-sided remote memory access ability of MPI-3, we implemented a task scoped global shared data object to enable sharing data sets on distributed memory systems through onesided RMA. With this shared data object, a user can implement and create a multi-threaded task running on either a single node or multiple nodes, and threads from distributed nodes can share and communicate through the global shared data object conveniently.

Chapter 4

Experiments and Results

To demonstrate the usability and performance of our framework and the implemented runtime system, we extened and ported a set of parallel applications based on the UTC framework. In this chapter, we show the details of the application development and use these applications to do experiments on real cluster platforms for analysis.

4.1 Benchmark Applications Development

4.1.1 Choosing Applications

There exist a number of different benchmark suites developed for testing and evaluating real systems. SPEC benchmarks [56] contain a series of benchmarks targeting different systems, such as SPEC Cloud for cloud computing, SPEC CPU for single CPU performance evaluation, SPEC MPI and SPEC OMP for testing parallel computing systems. However, these benchmarks are deeply tuned and programmed for their target systems and not convenient for reprogramming and porting. Also they are for commercial use and not open source projects. NPB [57] is a popular open source benchmark for high performance computing systems. It comprises several kernel applications both with sequential and parallel patterns. However, most of the programs are implemented in the Fortran language and limited to solving numerical computation problems. High Performance Linkpack (HPL) [58] is often used to evaluate the performance and efficiency of supercomputer systems. Besides benchmarks for traditional CPU systems, there are also many benchmarks developed for accelerator systems, especially GPUs, such as the Rodinia benchmarks [59].

All these benchmarks and included applications are usually designed to test various aspects

of the target system, such as the memory system, network system, power consumption, system scalability and so on. However, in this work, we not only focus on application performance or the speedup we can get, we also consider the programmability and usability when porting and reprogramming applications with our framework. Many full scale applications or benchmarks are well optimized and tuned programs. They are strictly bundled to the original programming models or methods, not easy and suitable to rewrite with a new programming method or language.

Table 4.1: Benchmark applications

Application	**Domain**
Image Rotation(Rotate)	Image Processing
Color Conversion(YUV)	Image Processing
Ray tracing(Raytrace)	Computer Graphics
MD5 Calculation(MD5)	Cryptography
Breadth First Search(BFS)	Graph Algorithm
Matrix Multiply(MM)	Linear Algebra
2D Heat Conduction(HC)	Linear Algebra
K-means Clustering(Kmeans)	Data Mining
N-body simulation(Nbody)	Astrophysics Simulation
Integral calculation(MC)	Linear Algebra

In this work, we take Starbench [60] and some other benchmarks [61, 59] as references and currently port ten applications to UTC as shown in Table 4.1. These applications covers various domains including image processing, linear algebra, simulation etc. Some applications are suitable for parallel processing and some are not. Each application usually contains a primary computing hot point which we implement as a task for parallel execution.

4.1.2 Application Implementations Summary

Our framework aims to facilitate users to develop parallel applications on heterogeneous computing platforms. Based on the task/conduit abstractions, low level task implementations are separated from application's high level structure, helping users to keep application programs modular and well organized for better portability. Here we will show some statistical information of programs' code line counts. We also implement these benchmark applications using other parallel programming methods such as Pthreads, OpenMP or MPI for tests and analysis.

Table 4.2: Application Total Code Line Information

Apps	Sequential	omp	Pthreads	mpi	CPU Task	sGPU Task	mGPU Task
Rotate	368	377	521	x	446	523	538
YUV	276	282	381	x	366	412	456
Raytrace	437	448	541	x	512	734	755
MD5	457	466	534	x	499	510	566
BFS	194	207	290	x	277	398	464
MM	132	x	x	220	280	355	393
HC	152	x	x	268	297	351	450
Kmeans	324	x	x	415	453	483	544
Nbody	556	x	x	616	646	888	954
MC	72	x	x	130	150	208	220

Table 4.2 shows the total number of lines of codes (LOC) for each application implemented through different programming methods. We realized three different versions through the UTC framework: *CPU Task*, in which we implemented a multi-threaded task running on a signle node or multiple nodes; *sGPU Task*, in which we implemented a GPU task that uses one GPU for acceleration on a single node; *mGPU Task*, in which we implemented a GPU task that make use of multiple GPUs for acceleration. Five of the ten applications (Rotate, YUV, Raytrace, MD5, BFS) are not suitable for running on multiple nodes, so they do not have an MPI version. For the remaining five applications, we programmed with MPI to run on cluster platforms and do not include OpenMP and Pthreads versions.

We can see that for different versions of parallel implementation, OpenMP versions usually have the fewest code lines. When using openMP, we only need to insert several compiler directives in sequential programs and the compiler helps implement and transform sequential programs to multi-threaded parallel programs. Our CPU Task version uses threads for parallel implementation, but we use explicit multiple threads to design and realize the parallel algorithms and therefore more code is needed. However, the CPU Task version has fewer code lines comparing to the explicit Pthreads version. This is mainly because the creation and management of multiple threads are all completed by our framework runtime, and users only focus on the algorithm implementation through our explicit multiple threads paradigm.

The CPU task version has slightly more code lines than the MPI version. This is mainly because the ported programs are multi-threaded on distributed platforms. In addition to coordinating threads on the same node, we also need to coordinate threads on different nodes. This is more like

MPI+thread programming and more code may be needed for synchronization and data management. Besides, these kernel applications usually do not contain complex or large amount of communication operations, so we may not be able to see the advantage of the one-sided RMA communication pattern through global shared data. For the GPU task versions, because we need to implement computation kernels through CUDA programming and mix CUDA programs with normal CPU programs, generally the GPU task versions have more code than traditional CPU programs. Also we use an Object Oriented programming approach to develop applications and these small applications may not reflect the benefits of OO programs.

Table 4.3: Task Code Line Information

Apps	**CPU Task**			**sGPU Task**			**mGPU Task**		
	Total	***Main***	***Task***	***Total***	***Main***	***Task***	***Total***	***Main***	***Task***
Rotate	446	94	132	523	110	170	538	114	189
YUV	366	93	68	412	108	109	456	112	126
Raytrace	512	104	245	734	125	326	755	128	341
MD5	499	95	237	510	114	367	566	118	392
BFS	277	105	95	398	129	156	464	134	209
MM	280	105	90	355	115	136	393	118	165
HC	297	109	134	351	116	154	450	122	212
Kmeans	453	119	178	483	130	206	544	136	252
Nbody	646	207	166	888	220	363	954	229	414
MC	150	78	55	208	86	90	220	88	105

Table 4.4: LOC Increment of Main Programs Between Different Versions

Apps	**CPU Task to sGPU Task**	**sGPU Task to mGPU Task**
Rotate	16	4
YUV	15	4
Raytrace	21	3
MD5	19	4
BFS	24	5
MM	10	3
HC	7	6
Kmeans	11	6
Nbody	13	9
MC	8	2

In Table 4.3 and Table 4.4, we elaborate the LOC information of each program's main

structure part, task implementation part and LOC changes of main part between differet versions. From CPU task version to single GPU task version and multiple GPUs task version, we can find there are few code line changes in a program's main part and the code line variations mainly derive from specific task implementations. This demonstrated that our high-level task and conduit design can help us implement well structured modular parallel applications, reducing the modifications when porting to different platforms.

4.2 Tests and Results with CPU/GPU Tasks

4.2.1 Test Platforms

In the following sections we show results from running on real systems with the benchmark applications. We ran these tests on the Discovery Cluster [62]. The cluster has various partitions with different hardware configurations. The partitions we have run programs on are:

- CPU partition: Each node has two 12 core CPUs with hyper-threading
- GPU-1 partition: Each node has a NVIDIA Tesla K20m GPU
- GPU-3 partition: 8 NVIDIA Tesla K80 GPUs on one node

The supported network backplane is 10 Gb/s TCP/IP ethernet. The underlying MPI implementation used is OpenMPI [18]. More platform hardware information can be found here [62].

4.2.2 Running on a Single Node

We first test different applications under the single node scenario. Targeting a single node multicore CPU platform, we create multi-threaded tasks for parallel processing. The tested CPU platform has two 12 core CPUs with hyper-threading, so we can run at most 48 threads without oversubscribing hardware resources. For the GPU platforms, if there is only one GPU on one node, we launch one thread in a GPU task to use the GPU for acceleration; If there are mutiple GPUs on one node, we will launch multiple threads and each thread controls and uses a unique GPU. As discussed above, we use five of the benchmark applications (Rotate, YUV, Raytrace, MD5, BFS) to test on single node platforms. We also ran the corresponded OpenMP implementation on the same platform. The speedup values in figures are normalized with log_2.

Figure 4.1 shows applications' total run time speedup compared to sequential programs with different number of threads for parallel processing. We can see that different applications

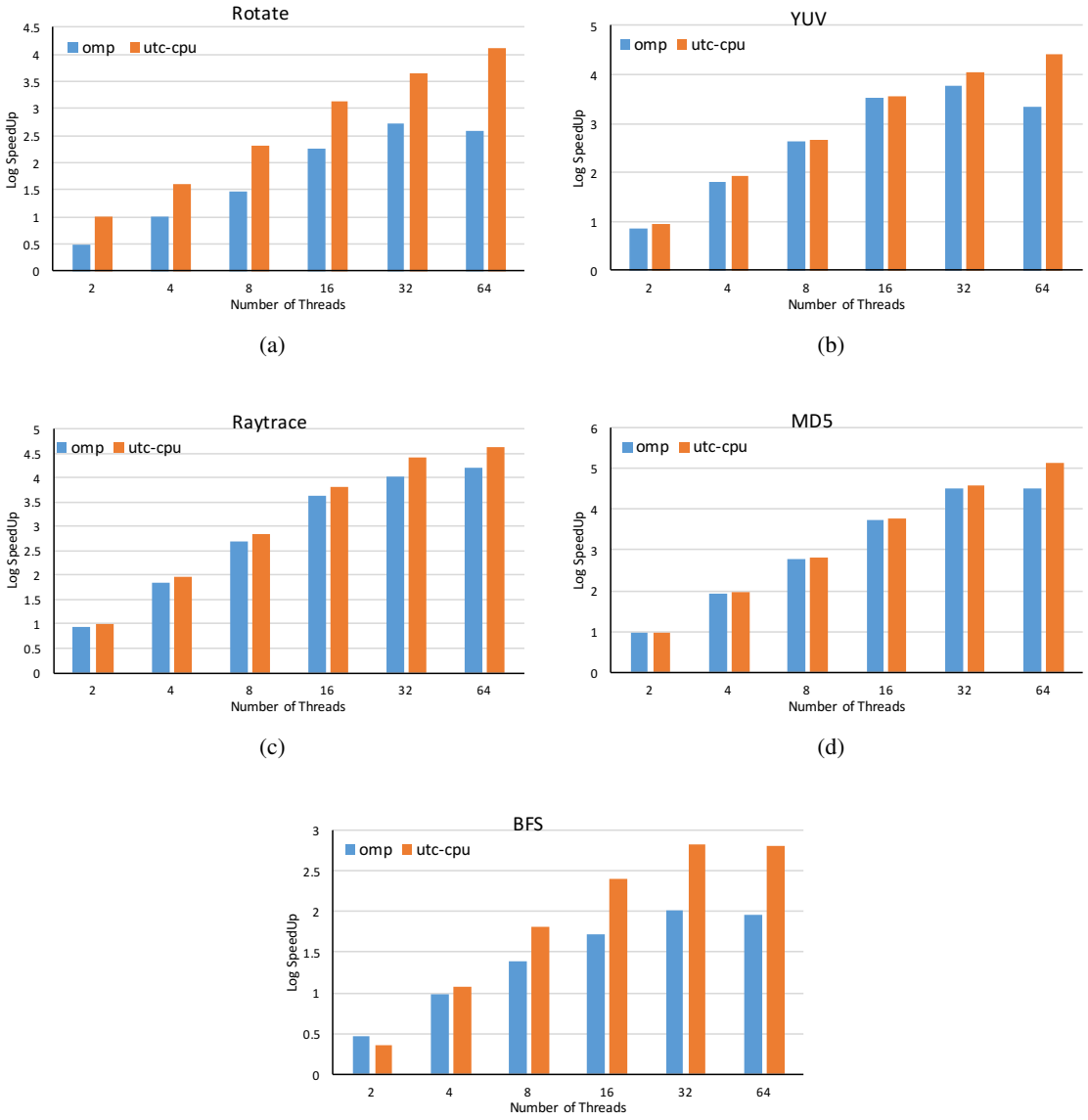


Figure 4.1: OpenMP and UTC CPU Task Execution Speedup to Sequential on Single Node

have different parallel processing performance. Some applications such as Raytrace and MD5 which contain little communication or synchronization procedures and can be highly parallelized, achieving speedup of more than 30 times in our test. But BFS which is data dependent and needs synchronization for every iteration only gets speedup up to 8 using all possible cores of one node.

We can usually get higher speedup using more threads when there are idle hardware resources. On our test platform, we have 24 cores with hyper-threading on one node and we are able to run as much as 48 threads in parallel. When we use 64 threads and oversubscribe the hardware resources, we get little or no speedup improvement compared with other cases. Also, the multi-threaded tasks have better performance than OpenMP implementations with the same number of threads. This is probably because OpenMP programs' parallelizations are done automatically by the compiler. Our framework runtime create required number of threads for the user to implement data parallelism explicitly in a multi-threaded task. The automatic parallelization of OpenMP may not be as good as explicit multi-threaded task implementations.

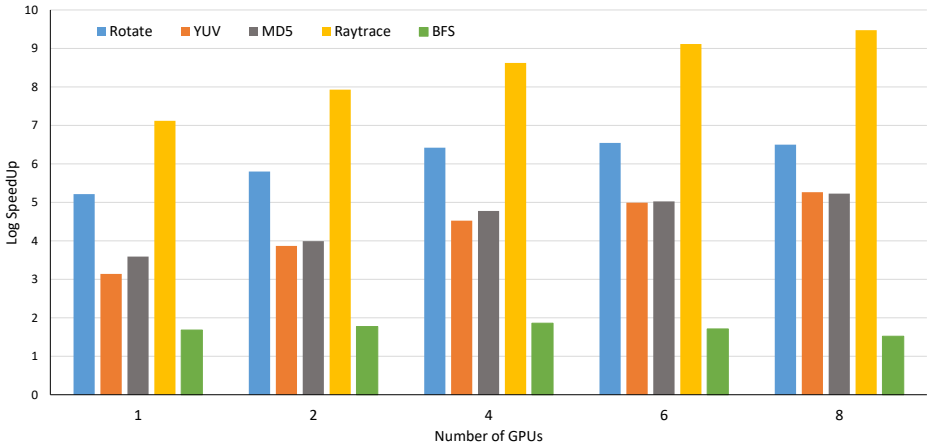


Figure 4.2: Execution Speedup with UTC GPU Task on Single Node

Figure 4.2 shows the speedup of using various number of GPUs on a single node compared to sequential programs. A GPU has independent memory for data access when executing GPU kernels, so the overhead of data movement between system main memory and GPU memory is a concern when porting applications to GPU. From the test results we find that when using more than four GPUs, we do not see much improvement for each application. With more GPUs, each GPU's computation workload decreases, so the data movement overhead as well as extra synchronizations for managing multiple GPUs may out weigh the benefit of using multiple GPUs for acceleration. Compared to Figure 4.1, we can see that one GPU's speedup performance is already better than or

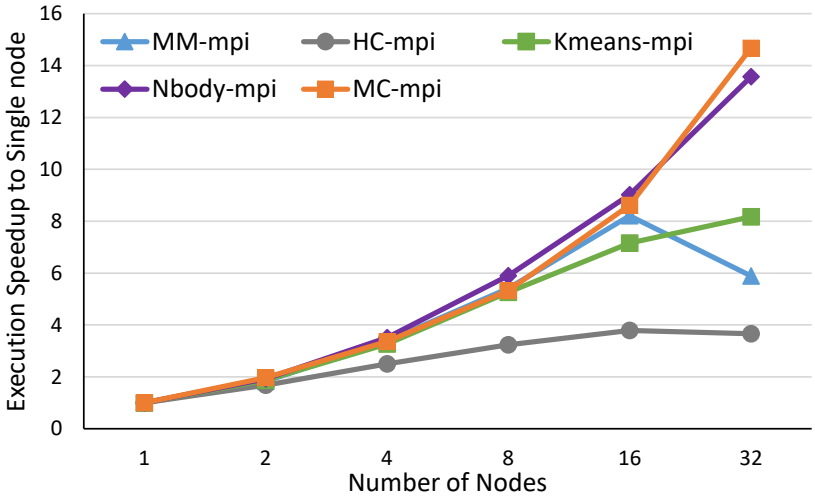
the same as running 64 threads on CPUs for each application except for BFS. So on platforms that are equipped with GPUs, user should leverage GPUs to accelerate applications to achieve overall better performance.

4.2.3 Running on a Cluster

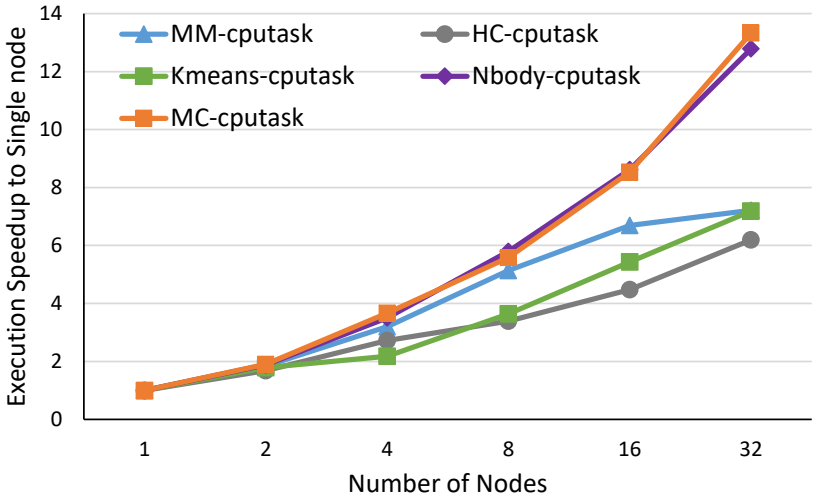
Next, we use another five applications (MM, HC, Kmeans, Nbody and MC) to test distributed cluster platforms. For CPU platforms, we run each application from 1 node to 32 nodes and on each node we run 24 parallel processes for MPI implementation and 24 task threads for CPU-task implementation. Each node of our test platform has two 12 core CPUs and when we run more than 24 processes on one node the intra-node MPI communication time would increase a lot. So we only run 24 parallel jobs on each node and ignore the hyper-threading effect. Figure 4.3a and Figure 4.3b show the strong scaling performance of multiple node execution for MPI implementations and CPU tasks respectively. The GPU platform we use has one GPU on one node and we scaled up to 16 nodes to make use of 16 GPUs for acceleration (see Figure 4.4).

From the test results we learn that MC and Nbody have better scaling performance than other applications when running on CPU cluster platforms due to the low amount of communication between parallel execution units. Especially in MC there is only one gather call needed. Other applications have more communication; in MM, kmeans and Nbody the total communication amount will increase along with more and more nodes being used. Further when more and more parallel jobs are created, each job's computational workload decreases and communication/synchronization time percentage of total execution time increases. This effect is even more critical while using GPUs. GPUs accelerate the computation a lot but other costs (inter-node data transfer, host/device data movement and synchronizations) may eliminate the acceleration benefits. We take MM and HC as examples to show the communication percentage variation with increasing number of nodes(Figure 4.5). GPU tasks run on at most 16 nodes, so there are no results for 32 nodes and host/device data transfer time is also included for the GPU case. We can see that the communication percentage of each program keeps growing and the GPU case is even worse. However, with our distributed multi-threaded task design and communication with global shared data, there is only one process on each node. This can help reduce inter-node and inter-process communications compared to pure MPI programs. This is why our scaling results of CPU tasks are better than the MPI tests.

Our test platforms use a common 10 Gb/s ethernet for network communication and our experiments are carried out under the real world scenario where there are lots of other users running



(a) MPI Test(24 processes per node)



(b) CPU Task Test(24 threads per node)

Figure 4.3: Test Showing Scaling Performance on CPU Cluster

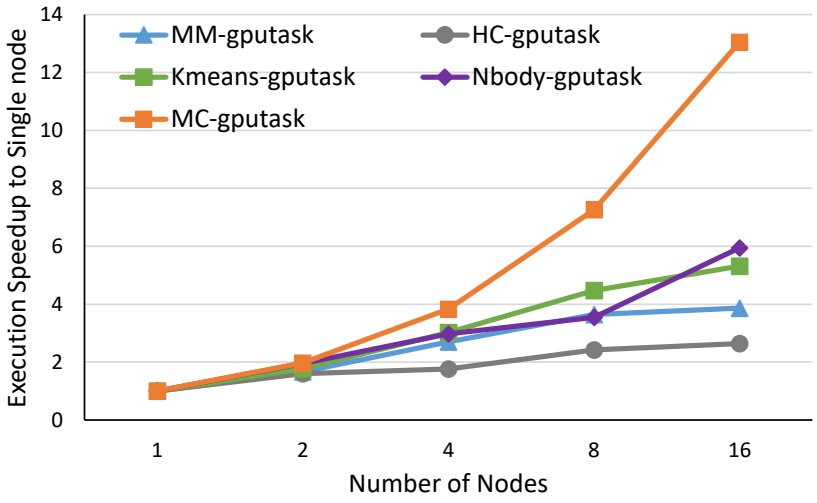


Figure 4.4: Test Showing Scaling Performance on GPU Cluster

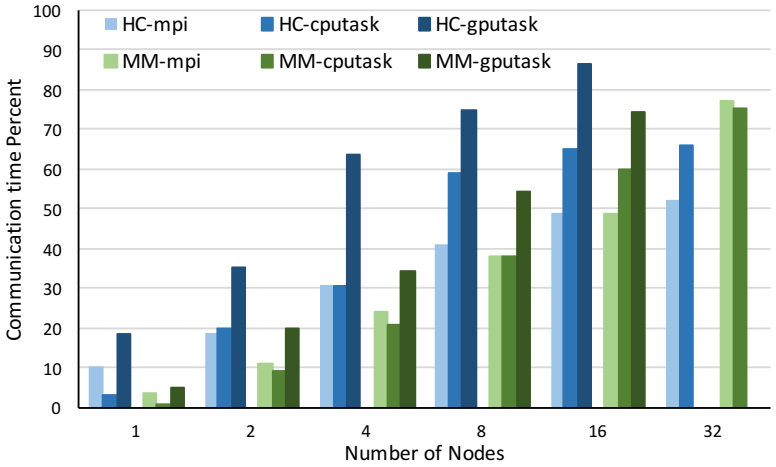


Figure 4.5: Percent of Total Run Time Costed by Communication

jobs on the cluster. So the low speed network and possibly heavy traffic and congestion may cause the applications to have a lower scaling performance. If testing on platforms with a high speed dedicated network, we should see better scaling behavior especially for GPU tasks.

4.3 Memory Exploration for GPU Tasks

4.3.1 GPU Tasks Performance

In Section 3.3 we have discussed there are three different types of memory (Pageable, Pinned, Unified) that one can use for host/device data transfer when implementing GPU tasks. Here we do experiments with the benchmark applications to check the effect of using different types of memory. Because we have implemented uniform interfaces for user to create memory blocks that will be used for host/device data movement, we only need to change the memory type options for GPU tasks to experience different memory without touching the existing task implementations. For each application, we prepare three different sizes of workload in our experiments, which are referred to as Small, Medium and Large (S, M, L) in following figures.

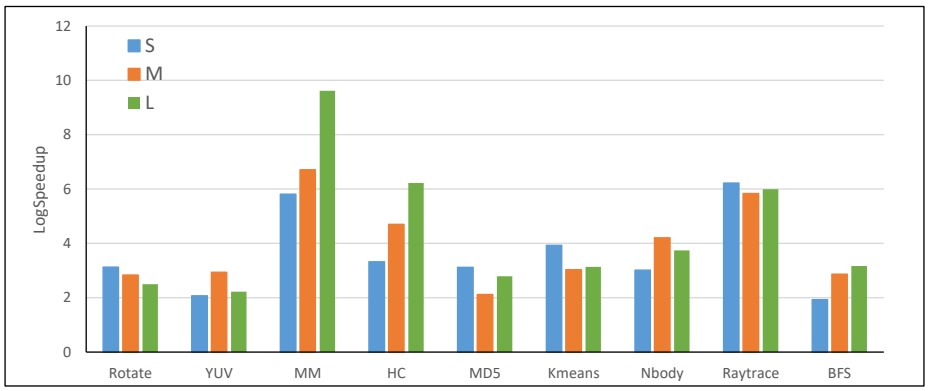


Figure 4.6: Applications speedup of different workloads on Tesla K20m with Pageable Memory

Figure 4.6 shows the general speedup of each application on a NVIDIA Tesla K20m GPU using pageable memory. Both time costs of data transfer between host/device and kernel execution are counted to calculate speedup. We can see that by using GPUs for acceleration we are able to get speedup, but outcomes of different applications vary a lot, ranging from 4 times faster for BFS to as large as 600 times for MM. Also, for each application, the speedup trend for different workload size is not consistent. There are many factors leading to this variation, such as GPU hardware characteristics, parallel algorithm design, device memory access pattern, host/device memory transfer and so on.

Therefore, great effort is required to develop a high performing implementation on a GPU platform. By designing and developing applications based on our high level task paradigm, we can keep a better program structure for porting and tuning applications on different platforms.

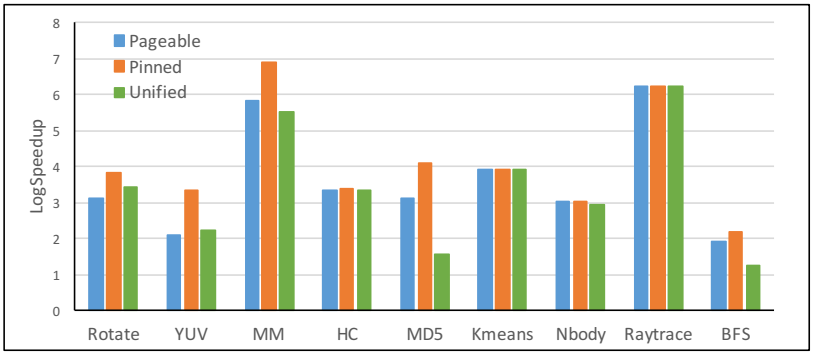
4.3.2 Pageable/Pinned/Unified Memory Usage

In this part we try each kind of memory with our benchmark applications. With the assistance of uniform memory interfaces, we only need to pass different memory type arguments to adjust and experience various types of memory for the application. Figure 4.7 shows the speedup results of kernel applications running on a Tesla K20m GPU.

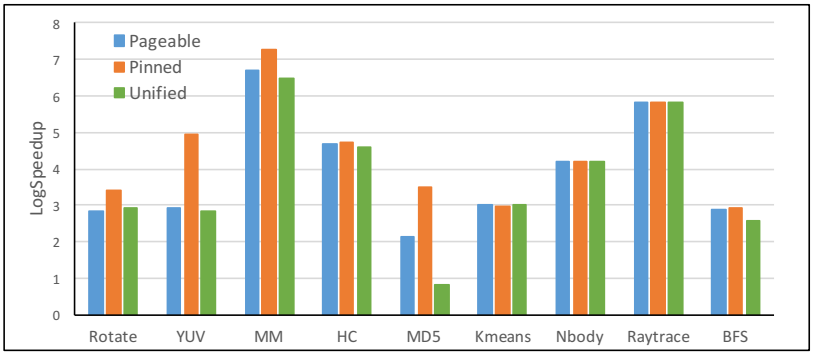
From the results we can see that, for Rotate, YUV and MD5, pinned memory performs better than pageable or unified memory. And for large workloads pinned memory is the best. For MM and BFS, using pinned memory is also preferable. However, for large workloads, the performance improvement by using pinned memory is not as good as for small and medium workloads. So considering the scarcity of pinned memory, pageable or unified may be a wise choice for applications with a large amount of data to process. For the rest of the applications, performance of the different types of memory are very close under all three workloads. If we want to save pinned memory usage we can use pageable memory for these applications, or use unified memory for program simplicity if you are not using our framework.

To further understand the characteristics of these kernel applications, we record time costs of both host/device communication and GPU computation. Figure 4.8 shows the this information as a percentage of total time using pageable memory for each application. We can see that host/device communication time of HC, K-Means, Nbody and Raytrace are much smaller compared to other applications. This is also why these four applications change less when adopting different types of memory. For MM and BFS, from small workloads to large workloads, we can see the time percentage of host/device communication reduced. So for large workloads, using pinned memory for them does not get much improvement.

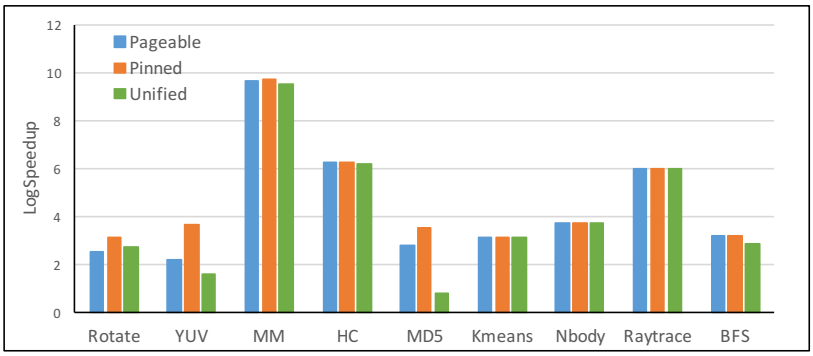
Figure 4.9 shows the host/device communication improvement when changing pageable memory to pinned memory for some of the benchmark applications. The pinned memory improvement in BFS reduces a lot for medium and large workloads mainly because the application implementation has an iterative structure, and during every iteration a flag value is transmitted to and from GPU device memory, but the amount of data transmitted is very small. This procedure does not dominate the communication time for small workloads, but for large workloads it is the primary part.



(a) Small Workload

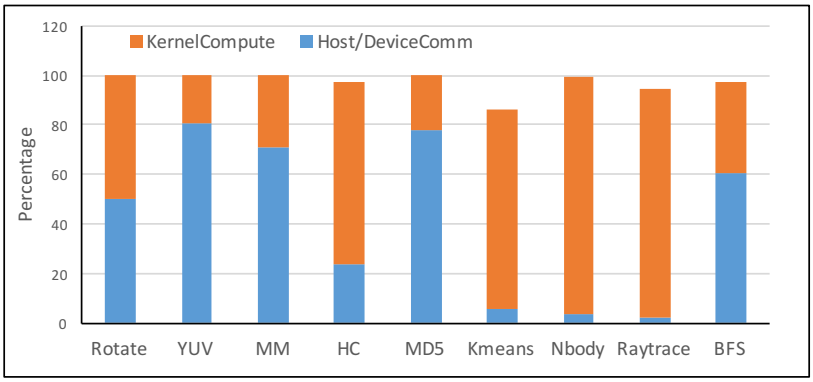


(b) Medium Workload

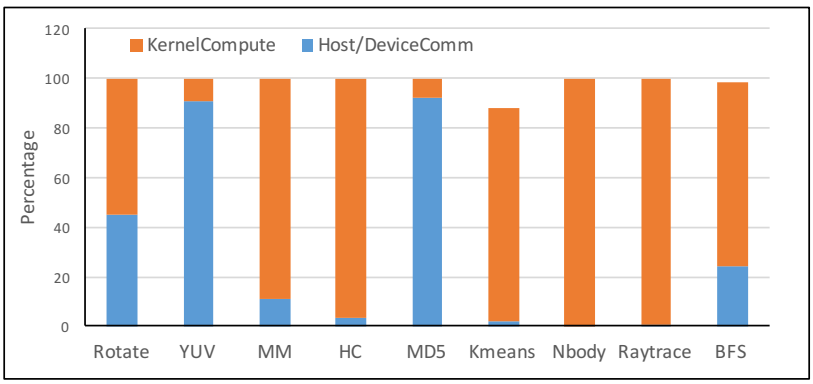


(c) Large Workload

Figure 4.7: Applications Speedup of Using Different Types of Memory on Tesla K20m



(a) Small Workload



(b) Large Workload

Figure 4.8: Computation/Communication Time Cost Percentage Using Pageable Memory

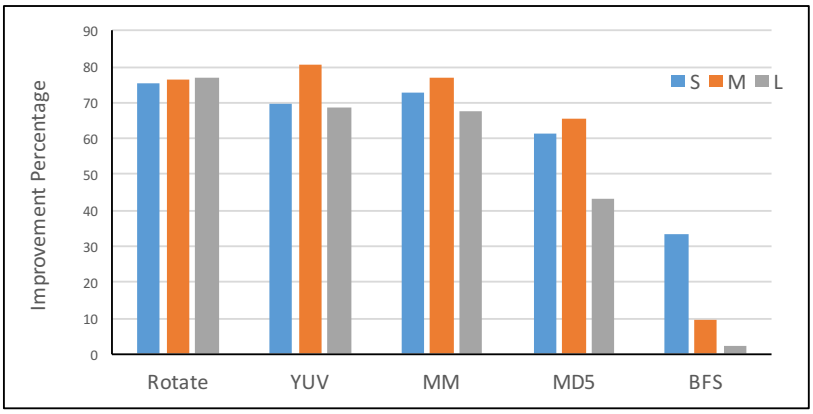


Figure 4.9: Host/Device Communication Improvement from Pageable to Pinned Memory

Due to the small size of transferred data, even with pinned memory, the bandwidth use ratio is still too low to get much advantage.

From these tests, we conclude that pinned memory can improve host/device communication performance significantly. For applications where memory transfer takes up a substantial amount of total runtime, using pinned memory can improve overall performance a lot. But memory transfer percentage may change with respect to the workload, like MM in our tests, so we should choose the memory type accordingly. Also, we cannot use pinned memory randomly. In some applications, choosing different memory types carefully depending on data size is a wise approach to improving overall performance. For choosing between different types of memory, with our interface it saves the user extra programming efforts, thus making the exploration of different design choices easier.

4.4 Multiple Tasks Use Case

In the above, we show experiments and results on various platforms with our benchmark applications. These applications usually contains one primary computation procedure, so they are kernel applications. Here we utilize our current kernel applications to compose a multiple task parallel application to study the usability of our framework for developing more comprehensive applications.

The application we constructed has three computation procedures: ray-tracing, YUV color convert and image rotation. The overall processing flow is shown as Figure 4.10. We will refer to this application as raytrace-yuv-rotate(RYR).

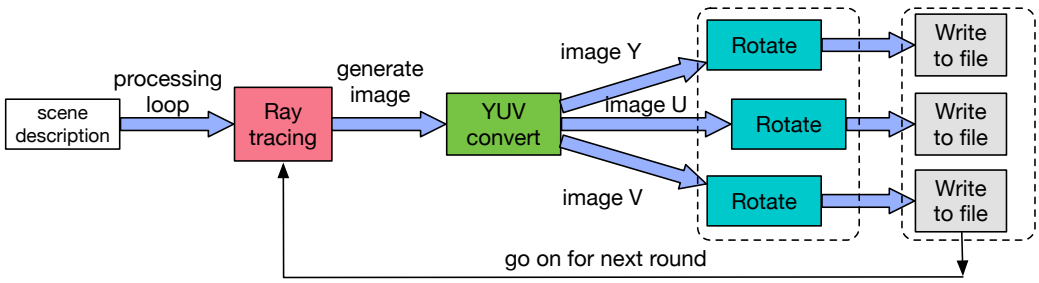


Figure 4.10: Work Flow for Raytrace-YUVconvert-Rotate(RYR)

We first read a scene description file to prepare the parameters for the ray tracing process. Then we enter the main processing loop: do ray tracing to generate image data sets; use generated image data to do YUV convert and produce three new images; then do rotate operation for each

image we get and write the final rotated image data sets to separate files on disk. We implement the sequential program, then test and calculate the processing rate(the number of frames completed per second) regarding different sizes of image for the application. The results are shown in Table 4.5. The bigger image we process, the more computation there is and the lower processing rate we will get. Also we can see that the ray tracing procedure takes up most of the total processing time.

Table 4.5: RYR Sequential Execution Information

Image size	**Processing rate (frames per second)**	**Time cost percentage in total time**			
		raytrace	*YUV*	*rotate*	*output*
512*512	1.137	78.02	3.51	9.81	8.65
1024*1024	0.291	78.91	3.42	9.69	7.98
2048*2048	0.077	76.29	3.64	10.75	9.32
4096*4096	0.020	76.23	3.71	11.00	9.07

In addition to implementing the sequential program, we have produced five different versions of the program under the UTC framework. Table 4.6 shows the differences of various implementations. *ryr-seq* is the sequential implementation and does not has any tasks. *ryr-v1* is a basic UTC version for this application. We create 4 tasks, one for each of the 4 different computation procedures and for each task we use one thread for execution. Between different tasks, we define necessary conduits for data communication.

In version *ryr-v2*, we launch 10 threads for the raytrace task to do parallel processing. Other tasks are the same as *ryr-v1*. Benefitting from our framework design, we only need to change the parameters in task creation to launch more threads for a task. As shown in Figure 4.10, the YUV procedure produces three new images for following processing, so in *ryr-v3* we create 3 rotate tasks and output tasks accordingly. Each pair is responsible for one image's processing. In *ryr-v4* and *ryr-v5*, we use a GPU task instead of a CPU task for the ray trace procedure. The main differences between *ryr-v4* and *ryr-v5* are: each YUV and rotate tasks in *ryr-v4* use 1 thread, while in *ryr-v5* we associate 3 threads for each of YUV and rotate tasks for parallel processing.

Figure 4.11 shows the processing rate of different implementations with image size of 1024*1024. We can see that from sequential implementation to multi-threaded tasks and GPU task implementation, we get higher and higher processing rates. For *ryr-v1*, we use four single threaded tasks to carry out the computations one by one. So the entire process is similar to the sequential program. But due to the pipeline execution pattern, described in Section 3.2.3, we can still get a

Table 4.6: Brief Info of Different RYR Implementations

Version	**Total codelines**	**Task Implementations of Each Procedure**			
		raytrace	*YUV*	*rotate*	*output*
ryr-seq	860	x	x	x	x
ryr-v1	1202(Basic task version)	1 CPUtask /1 thread	1 CPUtask /1 thread	1 CPUtask /1 thread	1 CPUtask /1 thread
ryr-v2	1202(no new code lines)	1 CPUtask /10 threads	1 CPUtask /1 thread	1 CPUtask /1 thread	1 CPUtask /1 thread
ryr-v3	1224(20 new code lines)	1 CPUtask /10 threads	1 CPUtask /1 thread	3 CPUtasks /3 threads	3 CPUtasks /3 threads
ryr-v4	1344(120 new code lines)	1 GPUtask /1 thread	1 CPUtask /1 thread	3 CPUtasks /3 threads	3 CPUtasks /3 threads
ryr-v5	1344(no new code lines)	1 GPUtask /1 thread	1 CPUtask /3 threads	3 CPUtasks /9 threads	3 CPUtasks /3 threads

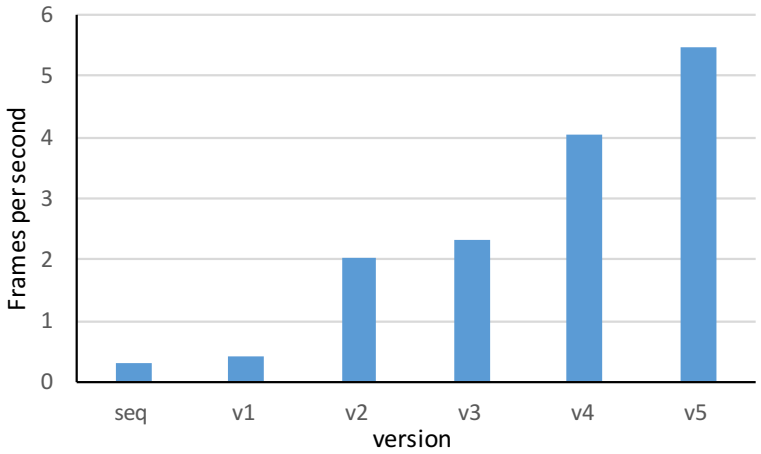


Figure 4.11: Processing Rate of Different RYR Implementations

better processing rate than sequential program.

In Table 4.7 we detail the execution time information for each version. We record the total run time as well as each computation procedure's execution time. For the sequential program, the total time is a summation of the four sub-procedures' time cost. For *ryr-v1*, the total run time is a little larger than raytrace's time, which indicate that other tasks' run time is hidden due to the pipeline

Table 4.7: RYR Execution Time of 50 Rounds(seconds)

Version	**Frames per second**	**Total run time**	**Raytrace time**	**YUV time**	**Rotate time**	**Output time**
ryr-seq	0.292	171.86	130.56	5.88	16.65	13.71
ryr-v1	0.396	126.24	123.24	15.02	22.82	13.71
ryr-v2	2.028	24.65	14.07	9.83	18.68	14.36
ryr-v3	2.340	21.37	14.27	10.09	7.71	5.73
ryr-v4	4.048	12.35	0.98	9.83	8.74	6.45
ryr-v5	5.475	9.13	0.96	4.68	5.25	7.82

execution effect. For *ryr-v2*, because raytrace task uses 10 threads for parallel processing, we can see that both the raytrace time and total time reduced a lot. In *ryr-v3*, we create 3 rotate tasks and 3 output tasks and they can execute in parallel, so the rotate time and output time keeps decreasing. But the total time does not reduce much. This is probably due to increased synchronization overhead with more and more threads. Also with more threads on the fly in the system, thread scheduling and cache behavior may have more effect on the overall parallel applications' performance.

For *ryr-v4* and *ryr-v5*, benefiting from the powerful parallel processing capability of GPU, raytrace time reduces greatly and the other three procedures become the primary time consuming part. Generally, through multi-threaded tasks and GPU tasks we are able to achieve nearly 20 times speedup compared to sequential execution. More important, with the help of our task/conduit design and framework runtime, a user can easily create multiple tasks to construct parallel applications. Through multi-threaded tasks and GPU tasks, users are able to explore task parallelism, data parallelism and pipeline parallelism at same time without much modification to the whole application implementation.

Chapter 5

Conclusion and Future work

5.1 Conclusion

In this dissertation I present a lightweight and flexible framework, Unified Tasks and Conduits(UTC), to facilitate parallel application development on heterogeneous computing platforms. In this framework, an application is constructed from high level task/conduit components, and each task can be either a sequential program, multi-threaded parallel program or kernel program that runs on CPUs or GPUs. When porting such a parallel application to utilize different computing resources on different platforms, the application's main structure can remain unchanged and only adopt appropriate versions of task implementations, reducing the development effort and improving program portability.

To support the framework, we have implemented a thin runtime system on a cluster platform, supporting the use of multicore CPUs and NVIDIA GPUs for tasks' parallel execution. To help develop GPU tasks, we provide a simple and effective interface for allocating and transferring data that enables users to flexibly choose pageable memory, pinned memory or unified memory for data communication between host main memory and GPU memory. To enable running tasks on distributed platforms, we leverage the one-sided remote memory access mechanism to implement a task scoped global shared data object. With this shared data object, a user can implement and create a multi-threaded task running on either a single node or multiple nodes, and threads from distributed nodes can share and communicate through the global shared data object conveniently.

We have ported and developed a set of benchmark applications based on our framework interfaces and runtime system. We use these applications to test on various platforms. The test results on a single node show that our multi-threaded task implementations have better performance

compared to OpenMP's automatic parallelization. By running applications from a single node up to 32 nodes, we find that the multi-threaded task implementations have better scaling performance compared to traditional, MPI-realized programs due to the hybrid shared memory mechanism. Further, through CUDA kernel integration we can create GPU tasks for each application to make use of GPUs of the tested platform to achieve better performance.

Generally, with the help of our framework and runtime implementation, users are able to develop parallel applications to explore task parallelism, data parallelism and pipeline parallelism flexibly and efficiently. Meanwhile, based on the high level task/conduit design, parallel applications maintain a well organized structure for improved portability and maintainability, reducing the effort of tuning application implementations for various platforms.

5.2 Future Work

Currently, our framework runtime system implementation focuses more on the correctness and functionality aspects and less effort has been dedicated to performance. There are several directions we can consider for future work:

1. **Threads and MPI process interaction**
   In our framework runtime design, we only launch one process on one node and make use of multiple threads for parallel processing on a single node. But the MPI library is process based. Using multiple threads in one MPI process and invoking MPI communication methods may suffer performance degradation and restrictions. Currently we make use of message tags to differentiate threads in the same process to avoid mismatch and errors. In future work, we may consider other approaches like [52] to deal with the usage of threads in an MPI environment and still keep good performance.

2. **User-level and other threads libraries**
   Currently, we make use of the Pthreads library to create and manage multiple threads. Pthreads is the Linux native threads library. The threads' creation, scheduling and management usually are OS kernel-level operations, and may be too heavy weight when thread count are very large. Using a user-level threads library is a preferred approach. With a user-level threads library we can change thread scheduling mechanisms dynamically and complete large amounts of thread management work with less overhead compared to system-level threads. There is some research about using a user-level threads library for better performance [38, 63]. Also, using

other threads library like Intel TBB [37] may get better performance when running on Intel CPUs. So providing support for other threads libraries in our framework runtime is another research direction.

3. **Conduit implement improvement**

   Conduits are used for data communication between different tasks. Our current conduit implementation supports both intra-node and inter-node communication of two tasks. However, a conduit instance is only used by two task instances. So it is a point-to-point conduit and for each different pairs of tasks, we need to create a dedicated conduit instance for them to transfer data sets. In future work, we will consider enhancing the conduits' capability, enabling a group of tasks to share the same conduit instance. Through shared conduits, multiple tasks can communicate with each other through a message publish-subscribe pattern. In this way, users can have more flexibility to design and implement applications with tasks/conduits.

4. **Support of other accelerators**

   Our current framework runtime implementation supports creating and running tasks on multicore CPUs and NVIDIA GPUs. Supporting other new accelerators such as FPGAs is also a direction of the future work. Benefiting from the power efficiency and parallel processing ability, FPGAs are more and more being used to accelerate applications. By extending the framework runtime, we will be able to interact with FPGA environment and launch tasks that are running on FPGAs for better performance.

5. **Application development**

   We have ported a series of applications with our framework and completed tests on cluster platforms. Next, we need to port and develop more applications (both small kernel applications and bigger real world applications) with our framework and test on various platforms to explore the performance bottleneck about our runtime implementations, demonstrating the advantages and improving our framework design and implementations.

Bibliography

[1] K. Rupp, "40 years of microprocessor trend data," https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/.

[2] NVIDIA, "Nvidia pg100 gpu whitepaper," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[3] "Knights landing (KNL): 2nd generation intel xeon phi," http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf.

[4] B. Lewis and D. J. Berg, *Multithreaded Programming with Pthreads*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[5] L. Dagum and R. Enon, "OpenMP: An industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," 2004.

[9] H. Kim, N. Bliss, R. Haney, J. Kepner, M. Marzilli, S. Mohindra, S. Sacco, G. Schrader, and E. Rutledge, "PVTOL: A high-level signal processing library for multicore processors," in *High Performance Embedded Computing Workshop*, 2007.

[10] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, no. 2, pp. 56–69, 2010.

[11] AMD, "Amd demonstrates breakthrough performance of next-generation zen processor core," http://www.amd.com/en-us/press-releases/Pages/zen-processor-core-2016aug18.aspx.

[12] G. Chrysos, "Intel® xeon phi coprocessor-the architecture," *Intel Whitepaper*, 2014.

[13] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin *et al.*, "Larrabee: a many-core x86 architecture for visual computing," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3. ACM, 2008, p. 18.

[14] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 13–24.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[16] N. T. Karonis, B. Toonen, and I. Foster, "Mpich-g2: A grid-enabled implementation of the message passing interface," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551–563, 2003.

[17] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar, "The mvapich project: Evolution and sustainability of an open source production quality mpi library for hpc," in *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Intl Conference on Supercomputing (WSSPE)*, 2013.

[18] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open mpi: A high-performance, heterogeneous mpi," in *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–9.

[19] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands *et al.*, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 2007, pp. 24–32.

[20] T. El-Ghazawi and L. Smith, "Upc: unified parallel c," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 27.

[21] K. Datta, D. Bonachea, and K. Yelick, "Titanium performance and potential: an npb experimental study," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2005, pp. 200–214.

[22] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.

[23] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 2.

[24] Nvidia. (2017) CUDA C programming guid. [Online]. Available: http://docs.nvidia.com/cuda/pdf/

[25] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[26] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870.

[27] R. Rabenseifner, "Hybrid parallel programming on HPC platforms," in *proceedings of the Fifth European Workshop on OpenMP, EWOMP*, vol. 3. Citeseer, 2003, pp. 185–194.

[28] X. Wu and V. Taylor, "Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 56–62, 2011.

[29] J. Brock, M. Leeser, and M. Niedre, "Adding support for GPUs to PVTOL: The Parallel Vector Tile Optimizing Library," in *14th Annual High Performance Embedded Computing Workshop*, 2010.

[30] D. Bonachea, "Gasnet specification, v1.1," Berkeley, CA, USA, 2002.

[31] H. Zhou, Y. Mhedheb, K. Idrees, C. W. Glass, J. Gracia, and K. Fürlinger, "Dart-mpi: an mpi-based implementation of a pgas runtime system," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 3.

[32] J. Daily, A. Vishnu, B. Palmer, H. van Dam, and D. Kerbyson, "On the suitability of mpi as a pgas runtime," in *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 2014, pp. 1–10.

[33] H. Zhou, K. Idrees, and J. Gracia, "Leveraging mpi-3 shared-memory extensions for efficient pgas runtime systems," in *European Conference on Parallel Processing*. Springer, 2015, pp. 373–384.

[34] J. R. Hammond, S. Ghosh, and B. M. Chapman, *Implementing OpenSHMEM Using MPI-3 One-Sided Communication*. Springer International Publishing, 2014, pp. 44–58.

[35] *OpenMP Application Program Interface, Version 4.0*, OpenMP Architecture Review Board, 2013.

[36] *User and Reference Guide for the Intel C++ Compiler 14.0*, Intel Corporation, 2014.

[37] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[38] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[39] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu, "The asynchronous partitioned global address space model," in *The First Workshop on Advances in Message Passing*, 2010, pp. 1–8.

[40] V. A. Saraswat, V. Sarkar, and C. von Praun, "X10: concurrent programming for modern architectures," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007, pp. 271–271.

[41] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "Clusterss: a task-based programming model for clusters," in *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 267–268.

[42] J. Labarta, "Starss: A programming model for the multicore era," in *PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany)*, 2010.

[43] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *European Conference on Parallel Processing*. Springer, 2011, pp. 555–566.

[44] D. Cunningham, R. Bordawekar, and V. Saraswat, "Gpu programming in a high level language: Compiling x10 to cuda," in *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, ser. X10 '11. New York, NY, USA: ACM, 2011, pp. 8:1–8:10.

[45] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 136–143.

[46] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU computing gems Jade edition*, vol. 2, pp. 359–371, 2011.

[47] R. Keryell, R. Reyes, and L. Howes, "Khronos sycl for opencl: A tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15. New York, NY, USA: ACM, 2015, pp. 24:1–24:1.

[48] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[49] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2. ACM, 1992, pp. 256–266.

[50] A. Foong, J. Fung, and D. Newell, "An in-depth analysis of the impact of processor affinity on network performance," in *Proceedings. 2004 12th IEEE International Conference on Networks (ICON 2004) (IEEE Cat. No.04EX955)*, vol. 1, Nov 2004, pp. 244–250 vol.1.

[51] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications,"

in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 180–186.

[52] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling MPI interoperability through flexible communication endpoints," in *Proceedings of the 20th European MPI Users' Group Meeting*. ACM, 2013, pp. 13–18.

[53] S. Sridharan, J. Dinan, and D. D. Kalamkar, "Enabling efficient multithreaded MPI communication through a library-based implementation of mpi endpoints," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 487–498.

[54] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in mpi-3," *ACM Trans. Parallel Comput.*, vol. 2, no. 2, pp. 9:1–9:26, Jun. 2015.

[55] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[56] Standard performance evaluation corporation (spec benchmarks). [Online]. Available: www.spec.org

[57] NASA parallel benchmark. [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[58] A. Petitet, "Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers," *http://www. netlib-. org/-benchmark/hpl/*, 2004.

[59] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[60] M. Andersch, B. Juurlink, and C. C. Chi, "A benchmark suite for evaluating parallel programming models," in *Proceedings of Workshop on Parallel Systems and Algorithms (PARS), Gesellschaft Fur Informatik*, vol. 28, 2013, pp. 1–6.

[61] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[62] Overview of discovery cluster. [Online]. Available: https://www.northeastern.edu/rc/?page_id=27

[63] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castello, D. Genet, T. Herault, S. Iwasaki, P. Jindal, S. Kale, S. KRISHNAMOORTHY, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2017.

Appendix A

Information of Data Sets Used for Application Tests

In Chapter 4 we have done various experiments with our benchmark applications. Different sizes of computation workload produce different test results for each application. Here we show the detailed information of data sets we used in our application tests respected to each tested results.

Table A.1: Data sets used for Figure 4.1

Application	Data Sets Description	Data Type
Rotate	8192 * 8192 pixels	Float
YUV	8192 * 16384 pixels	Float
Raytrace	2048 * 2048 pixels	Float
MD5	512K buffers with 16K bytes per buffer	Char
BFS	8M vertices with about 40M edges	Int

Table A.2: Data sets used for Figure 4.2

Application	Data Sets Description	Data Type
Rotate	8192 * 8192 pixels	Float
YUV	8192 * 16384 pixels	Float
Raytrace	4096 * 4096 pixels	Float
MD5	512K buffers with 16K bytes per buffer	Char
BFS	4M vertices with about 20M edges	Int

Table A.3: Data sets used for Figure 4.3

Application	Data Sets Description	Data Type
MM	7680*8192 and 8192*7680 matrices	Float
HC	4096*92160 matrix	Float
Kmeans	6144000 objects with 16 attributes per object	Float
Nbody	61440 bodies	Double
MC	15360M iterations	Double

Table A.4: Data sets used for Figure 4.4

Application	Data Sets Description	Data Type
MM	16K*16K and 16K*16K matrices	Float
HC	8K*64K matrix	Float
Kmeans	8M objects with 16 attributes per object	Float
Nbody	128K bodies	Double
MC	32G iterations	Double

Table A.5: Small Workload for All Tests in Chapter 4.3

Application	Data Sets Description	Data Type
Rotate	320*200 pixels	Float
YUV	320*200 pixels	Float
MM	256*256 and 256*256 matrices	Float
HC	128*128 matrix	float
MD5	2K buffers with 4K bytes per buffer	Char
Kmeans	8K objects with 16 features per object	Float
Nbody	1k bodies	Double
Raytrace	128*128 pixels	Float
BFS	64K vertices with about 160K edges	Int

Table A.6: Medium Workload for All Tests in Chapter 4.3

Application	Data Sets Description	Data Type
Rotate	2K*1K pixels	Float
YUV	2K*1K pixels	Float
MM	1K*1K and 1K*1K matrices	Float
HC	512*512 matrix	float
MD5	32K buffers with 8K bytes per buffer	Char
Kmeans	128K objects with 16 features per object	Float
Nbody	8k bodies	Double
Raytrace	512*512 pixels	Float
BFS	27K vertices with about 740K edges	Int

Table A.7: Large Workload for All Tests in Chapter 4.3

Application	Data Sets Description	Data Type
Rotate	8K*3K pixels	Float
YUV	8K*3K pixels	Float
MM	4K*4K and 4K*4K matrices	Float
HC	2K*2K matrix	float
MD5	256K buffers with 16K bytes per buffer	Char
Kmeans	2M objects with 16 features per object	Float
Nbody	32k bodies	Double
Raytrace	2K*2K pixels	Float
BFS	1M vertices with about 5M edges	Int

Appendix B

Application Example and Basic Framework Interfaces

B.1 Application Code Example

Here we show an application's code snippet using UTC framework. The application is the RYR(Raytace-YUV-Rotate) application which we used for test in Chapter 4.4 and we only list the main part of the application.

```
1
2 /*
3  * ryr_main.cc
4  *
5  * Raytrace-YUV-Rotate application
6  *
7  */
8 #include "../../common/helper_getopt.h"
9 #include "../../common/helper_timer.h"
10 #include "../../common/helper_printtime.h"
11 #include "Utc.h"
12 #include "UtcGpu.h"
13 #include "common.h"
14 #include <fstream>
15 #include <iostream>
16 #include <cstdlib>
17 #include <cstdio>
18 #include <cmath>
```

```
19 #include <cstring>
20 #include <cstdint>
21
22 using namespace iUtc;
23
24 /*
25  * task implementations for different tasks
26  */
27 #include "task.h"
28 #include "cpu_task/ray_task.h"
29 #include "cpu_task/yuv_task.h"
30 #include "cpu_task/rotate_task.h"
31 #include "cpu_task/output_task.h"
32 #include "gpu_task/c-ray_task_sgpu.h"
33
34 #define MAX_THREADS 64
35 FTYPE aspect = 1.333333;
36
37 int main(int argc, char** argv){
38   bool printTime = false;
39   char* infile_path = NULL;
40   int xres=800;
41   int yres=600;
42   int rays_per_pixel=1;
43   int loop = 100;
44
45   int nthreads=1;
46   int nprocess=1;
47
48   MemType memtype = MemType::pageable;
49   int mtype = 0;
50
51   /* initialize UTC context */
52   UtcContext &ctx = UtcContext::getContext(argc, argv);
53
54   /* Parse command line options */
55   int     opt;
56   extern char   *optarg;
57   extern int     optind;
58   while ( (opt=getopt(argc,argv,"w:h:i:l:vt:p:m:"))!= EOF) {
```

```
59      switch (opt) {
60        case 'v': printTime = true;
61            break;
62        case 't': nthreads=atoi(optarg);
63            break;
64        case 'p': nprocess = atoi(optarg);
65            break;
66        case 'm': mtype = atoi(optarg);
67            break;
68        case 'i': infile_path=optarg;
69            break;
70        case 'w': xres = atoi(optarg);
71            break;
72        case 'h': yres = atoi(optarg);
73            break;
74        case 'l': loop = atoi(optarg);
75            break;
76        case ':':
77          std::cerr<<"Option -"<<(char)optopt<<" requires an operand\n"<<std::endl
      ;
78          break;
79        case '?':
80          std::cerr<<"Unrecognized option: -"<<(char)optopt<<std::endl;
81          break;
82        default:
83            break;
84      }
85    }
86    int procs = ctx.numProcs();
87    int myproc = ctx.getProcRank();
88    if(nprocess != procs){
89      std::cerr<<"process number not match with arguments '-p' !!!\n";
90      return 1;
91    }
92    // used for GPU tasks to configure memory for host/device copy
93    if(mtype==0)
94      memtype = MemType::pageable;
95    else if(mtype==1)
96      memtype = MemType::pinned;
97    else if(mtype ==2)
```

```
98      memtype = MemType::unified;
99    else
100     std::cerr<<"wrong memory type for -m !!!"<<std::endl;
101
102   /*
103    * read input scene file
104    */
105   global_vars g_vars;
106   g_vars.xres = xres;
107   g_vars.yres = yres;
108   g_vars.rays_per_pixel = rays_per_pixel;
109   g_vars.aspect = aspect;
110   sphere_array_t obj_array_for_gpu;
111   vec3_t lights[MAX_LIGHTS];
112   Task<SceneInit> sceneConfig(ProcList(0));
113   sceneConfig.run(infile_path, &obj_array_for_gpu, &g_vars, lights);
114   sceneConfig.wait();
115   sphere2_t *obj_array_for_cpu = new sphere2_t[g_vars.obj_count];
116   for(int i = 0; i < g_vars.obj_count; i++){
117     obj_array_for_cpu[i].mat = obj_array_for_gpu.mat[i];
118     obj_array_for_cpu[i].pos = obj_array_for_gpu.pos[i];
119     obj_array_for_cpu[i].rad = obj_array_for_gpu.rad[i];
120   }
121   uint32_t *pixels_array = (uint32_t*)malloc(xres * yres * sizeof(uint32_t) *
        loop);
122
123
124   /*
125    * Create necessary task/conduit instances
126    * Execute tasks
127    * Wait for tasks complete
128    */
129
130   /*
131    * Raytrace task
132    *
133    *  running on node 0 using 10 threads
134    */
135   double ray_runtime[MAX_THREADS][3];
136   int plist1[10] = {0,0,0,0,0,0,0,0,0,0};
```

```
137  Task<crayCPUWorker> raytrace(ProcList(10, plist1), TaskType::cpu_task);
138  //Task<craySGPU> raytrace(ProcList(1, plist1), TaskType::gpu_task, memtype);
139
140  /*
141   * YUV task
142   *
143   *  running on node 0 using 3 threads
144   */
145  double yuv_runtime[MAX_THREADS][3];
146  int plist2[3] = {0,0,0};
147  Task<YUVconvertCPUWorker> yuv(ProcList(3, plist2), TaskType::cpu_task);
148  /*
149   * conduit between raytrace task and YUV task
150   */
151  Conduit cdt1(&raytrace, &yuv);
152
153  /*
154   * rotate task
155   *
156   *  tree rotate task, each use 3 threads for running
157   *
158   *  each rotate task has a conduit to YUV task
159   */
160  double rotate_runtime[MAX_THREADS][3];
161  int plist3[3] = {0,0,0};
162  Task<RotateCPUWorker> rotate(ProcList(3, plist3), TaskType::cpu_task);
163  Conduit cdt2(&yuv, &rotate);
164  Task<RotateCPUWorker> rotate2(ProcList(3, plist3), TaskType::cpu_task);
165  Conduit cdt22(&yuv, &rotate2);
166  Task<RotateCPUWorker> rotate3(ProcList(3, plist3), TaskType::cpu_task);
167  Conduit cdt23(&yuv, &rotate3);
168
169  /*
170   * output task
171   *
172   *  tree output task, each use 1 threads for running
173   *
174   *  each has a conduit to one rotate task accordingly
175   *
176   */
```

```
177  double output_runtime[MAX_THREADS][3];
178  int plist4[1] = {0};
179  Task<OutputWorker> output(ProcList(1, plist4), TaskType::cpu_task);
180  Conduit cdt3(&rotate, &output);
181  Task<OutputWorker> output2(ProcList(1, plist4), TaskType::cpu_task);
182  Conduit cdt32(&rotate2, &output2);
183  Task<OutputWorker> output3(ProcList(1, plist4), TaskType::cpu_task);
184  Conduit cdt33(&rotate3, &output3);
185
186  /*
187   * initialize all defined tasks
188   */
189  raytrace.init(g_vars, obj_array_for_cpu, pixels_array, lights, &cdt1);
190  //raytrace.init(g_vars, obj_array_for_gpu, pixels_array, lights, &cdt1);
191
192  std::vector<Conduit*> cdts;
193  cdts.push_back(&cdt2);
194  cdts.push_back(&cdt22);
195  cdts.push_back(&cdt23);
196  yuv.init(xres, yres, 5, loop, &cdt1, cdts);
197
198  rotate.init(xres, yres, &cdt2, &cdt3);
199  rotate2.init(xres, yres, &cdt22, &cdt32);
200  rotate3.init(xres, yres, &cdt23, &cdt33);
201
202  Timer timer;
203  timer.start();
204
205  /*
206   * run all the tasks
207   *
208   */
209  raytrace.run(ray_runtime, loop, 1);
210  //raytrace.run(ray_runtime, memtype, loop);
211
212  yuv.run(yuv_runtime);
213
214  rotate.run(rotate_runtime, loop);
215
216  double tmptime[MAX_THREADS][3];
```

```
217   rotate2.run(tmptime, loop);
218
219   double tmptime2[MAX_THREADS][3];
220   rotate3.run(tmptime2, loop);
221
222   output.run(loop, &cdt3, output_runtime, 0);
223
224   double tmptime3[MAX_THREADS][3];
225   output2.run(loop, &cdt32, tmptime3, 1);
226
227   double tmptime4[MAX_THREADS][3];
228   output3.run(loop, &cdt33, tmptime4, 2);
229
230   /*
231    * wait for task finishing computation
232    */
233   raytrace.wait();
234   //std::cout<<"ray finish wait"<<std::endl;
235   yuv.wait();
236   //std::cout<<"yuv finish wait"<<std::endl;
237   rotate.wait();
238   rotate2.wait();
239   rotate3.wait();
240   //std::cout<<"rotate finish wait"<<std::endl;
241   output.wait();
242   output2.wait();
243   output3.wait();
244   //std::cout<<"output finish wait"<<std::endl;
245
246
247   /*
248    * print timing info
249    */
250   ctx.Barrier();
251   double totaltime = timer.stop();
252
253   if(myproc == 0){
254     std::cout<<"Test complete !!!"<<std::endl;
255     std::cout<<"\tTotal time: "<<std::fixed<<std::setprecision(4)<<totaltime<<"(
      s)"<<std::endl;
```

```
256     std::cout<<"\traytrace time: "<<std::fixed<<std::setprecision(4)<<
        ray_runtime[0][1]<<"(s)"<<std::endl;
257     std::cout<<"\tyuv time: "<<std::fixed<<std::setprecision(4)<<yuv_runtime
        [0][1]<<"(s)"<<std::endl;
258     std::cout<<"\trotate time: "<<std::fixed<<std::setprecision(4)<<
        rotate_runtime[0][1]<<"(s)"<<std::endl;
259     std::cout<<"\toutput time: "<<std::fixed<<std::setprecision(4)<<
        output_runtime[0][1]<<"(s)"<<std::endl;
260   }
261
262   ctx.Barrier();
263   return 0;
264 }
```

B.2 Basic classes/methods in Framework Runtime Implementation

```
1
2 /*
3  * initialize UTC context
4  */
5 static UtcContext& UtcContext::getContext(int &argc, char** &argv);
6
7
8 /*
9  * task constructors and methods
10 */
11 template<class T>
12 class Task: public TaskBase{
13 public:
14   // constructors
15   Task();
16   Task(ProcList rList);
17   Task(ProcList rList, TaskType tType);
18   Task(T* userTaskObj, ProcList rList, TaskType tType);
19   Task(std::string name);
20   Task( std::string name , ProcList rList);
21   Task(std::string name, ProcList rList, TaskType tType);
22   Task(std::string name, ProcList rList, TaskType tType, long shmemSize);
23   Task(ProcList rList, TaskType tType, long shmemSize);
```

```
24    Task(ProcList rList, TaskType tType, int bind_mode);
25    Task(ProcList rList, TaskType tType, long shmemSize, int bind_mode);
26
27    // task init method
28    template <typename... Args>
29    void init(Args... args);
30
31    // task run method
32    template <typename... Args>
33    void run(Args... args);
34
35    // task wait method
36    void wait();
37
38    // task finish method
39    void finish();
40
41 };
42
43 /*
44  * conduit constructors and methods
45  */
46 class Conduit{
47 public:
48    // constructors
49    Conduit(TaskBase* srctask, TaskBase* dsttask);
50
51    // write mthods
52    int Write(void* DataPtr, DataSize_t DataSize, int tag);
53    int WriteBy(ThreadRank_t thread, void* DataPtr, DataSize_t DataSize, int tag);
54
55    // read methods
56    int Read(void* DataPtr, DataSize_t DataSize, int tag);
57    int ReadBy(ThreadRank_t thread, void* DataPtr, DataSize_t DataSize, int tag);
58
59 };
60
61
62 /*
63  * uniform GPU data interface used in GPU task implementation
```

```
64  */
65 template<typename T>
66 class GpuData{
67 public:
68   // constructors
69   GpuData(MemType memtype = MemType::pageable);
70   GpuData(unsigned long size, MemType memtype = MemType::pageable);
71   GpuData(unsigned long size_x, unsigned long size_y, MemType memtype = MemType
       ::pageable);
72   GpuData(unsigned long size_x, unsigned long size_y, unsigned long size_z,
       MemType memtype = MemType::pageable);
73
74   // get memory address
75   T *get();
76   T *getH();
77   T *getD();
78
79   // sync host/device memory
80   T *sync();
81   T *syncH();
82   T *syncD();
83
84   // put data to inside buffer
85   void put(const T *src);
86   void putH(const T *src);
87   void putD(const T *src);
88
89   // fetch data from inside buffer
90   void fetch(T *dst);
91   void fetchH(T *dst);
92   void fetchD(T *dst);
93 };
94
95
96 /*
97  * global scoped data and related methods
98  */
99 template<typename T>
100 class GlobalScopedData: public GlobalScopedDataBase{
101 public:
```

```
102   // constructors
103   GlobalScopedData(long size=1);
104
105   // get datablock's base address
106   T *getPtr();
107
108   // single element load/store from local
109   T load(int index=0);
110   void store(T value, int index=0);
111
112   // datablock load/store from local
113   int loadblock(T* dst, int startIdx, int blocks);
114   int storeblock(T* src, int startIdx, int blocks);
115
116   // single element load/store from remote
117   T rload(int remotePE, int index=0);
118   void rstore(int remotePE, T value, int index=0);
119
120   // datablock load/store from remote
121   int rloadblock(int remotePE, T* dst, int startIdx, int blocks);
122   int rstoreblock(int remotePE, T* src, int startIdx, int blocks);
123
124   // wait for remote operation complete
125   void quiet();
126
127   // synchronization method
128   void barrier();
129
130 };
131
132
133 /*
134  * base interface class for user to implement when user want to
135  * implement specific tasks for their needs
136  */
137 class UserTaskBase{
138 public:
139   /* necessary methods must be implemented */
140   virtual void initImpl();
141
```

```
142   virtual void runImpl();
143
144   /* pre-defined useful data members*/
145   static thread_local int __localThreadId;
146   static thread_local int __globalThreadId;
147   static thread_local int __processIdInWorld;
148   static thread_local int __processIdInGroup;
149   int __numLocalThreads=0;
150   int __numGlobalThreads=0;
151   int __numWorldProcesses=0;
152   int __numGroupProcesses=0;
153
154 };
```