A Framework for Developing Parallel Applications with high level Tasks on Heterogeneous Platforms

Chao Liu
Northeastern University
liu.chao@husky.neu.edu

Miriam Leeser
Northeastern University
mel@coe.neu.edu

Abstract

Traditional widely used parallel programming models and methods focus on data distribution and are suitable for implementing data parallelism. They lack the abstraction of task parallelism and make it inconvenient to separate the applications' high level structure from low level implementation and execution. To improve this, we propose a parallel programming framework based on the tasks and conduits (TNC) model. In this framework, we provide tasks and conduits as the basic components to construct applications at a higher level. Users can easily implement coarse-grained task parallelism with multiple tasks running concurrently. When running on different platforms, the application main structure can stay the same and only adapt task implementations based on the target platforms, improving maintenance and portability of parallel programs. For a single task, we provide multiple levels of shared memory concepts, allowing users to implement fine grained data parallelism through groups of threads across multiple nodes. This provides users a flexible and efficient means to implement parallel applications. By extending the framework runtime system, it is able to launch and run GPU tasks to make use of GPUs for acceleration. The support of both CPU tasks and GPU tasks helps users develop and run parallel applications on heterogeneous platforms. To demonstrate the use of our framework, we tested it with some kernel applications. The results show that the applications' performance using our framework is comparable to traditional programming methods. Further, with the use of GPU tasks, we can easily adjust the program to leverage GPUs for acceleration. In our tests, a single GPU's performance is comparable to a 4 node multicore CPU cluster.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features–Concurrent programming structures, Frameworks

General Terms Design, Languages

Keywords Tasks&Conduits, Task parallelism, Shared memory, GPU, Accelerator

1. Introduction

With the prevalence of multicore and manycore processors, parallel programming and parallel application development has become more and more important. For single node systems, shared memory programming methods such as Pthreads or OpenMP are widely adopted. For cluster systems, MPI is the most popular programming method. Also, the Partitioned Global Address Space (PGAS) model [1] is attracting more and more research interest. It extends the shared memory concept to a distributed memory environment, allowing a group of processes to share a virtual global memory space across multiple nodes and make use of one-sided communication mechanisms to access remote data. Targeting heterogeneous parallel computing platforms, which have both CPUs and accelerators, novel programming languages or models have been introduced to develop programs running on accelerators. OpenCL [2] and CUDA [3] are two primary programming methods for producing parallel applications on GPUs. In addition to programming with a single method, users often combine different programming methods together, such as MPI+OpenMP or MPI+CUDA.

The above programming methods usually focus on data distribution and providing high performance data communication. They are suitable for implementing data parallelism in an application, but they lack the abstraction of high level task parallelism. 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.

To address these problems, we propose a lightweight unified tasks and conduits (UTC) parallel programming framework based on the tasks and conduits (TNC) model. In the TNC model, tasks are the abstraction of computation workloads while conduits are the abstraction of data transfer between different tasks. TNCallows a user to express explicit task parallelism easily. Using this model, application development includes two levels. 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. 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.

PMAM'17, February 04-08, 2017, Austin, TX, USA
Copyright © 2017 ACM 978-1-4503-4883-6/17/02. . . $15.00
DOI: http://dx.doi.org/10.1145/3026937.3026946

Using the TNC model as a high level abstraction for applications was first presented in [4]. Targeting signal processing applications, the authors provided a Parallel Vector Tile Optimizing Library (PVTOL) to help developers create portable parallel programs on multicore platforms. However, the PVTOL library is mainly used for developing signal processing programs, and TNC is not implemented as a single layer for easy extension. This restricted the usage of this model.

We propose the UTC framework based on the concepts of TNC and provide features to help develop and run parallel application on heterogeneous platforms. In the framework we leverage the PGAS programming model's global shared memory feature to improve tasks' ability to perform more complex parallel algorithms. A task uses a group of threads for parallel execution. These threads can be on a single node or multiple nodes. Threads on the same node can work with traditional shared memory, and threads on different nodes will work with global shared memory, still sharing data with each other. So in the UTC framework we provide a hybrid threads + PGAS programming method for users to implement parallel programs within a task, improving the usability of tasks. However, data sharing and hybrid threads + PGAS is only applied to single tasks, it is not available between different tasks. Different tasks do not share data with each other, they perform explicitly communication through conduits. Targeting heterogeneous platforms, our framework is not constrained to only running parallel applications on CPU platforms; it supports both CPU tasks and accelerator tasks. Various types of task are created and used in a uniform way so that an application's main structure remains the same and is constructed of tasks. Only the task implementation changes when porting the program to make use of different computing resources on different target platforms.

The contributions of this paper are:

- Providing uniform task and conduit components for constructing parallel applications on heterogeneous platforms through the UTC framework, isolating application high level structure from low level task implementation and execution, thus improving application maintenance and portability.
- Supporting both CPU and GPU tasks and allowing different tasks to run in parallel in a MPMD manner and communicate through conduits to express task parallelism. The global shared data object enables a single task to make use of hybrid thread and PGAS method to implement data parallelism with multiple threads.
- Completing pipeline execution of multiple tasks easily and overlapping communication and computation within a multi-threaded task to improve performance through the framework.
- Implementing four kernel applications to test and demonstrate the use of this framework.

The paper is organized as follows: Section 2 shows the design of the framework. In Section 3 we discuss the framework runtime system implementation and extension for GPU platforms. Test results and analysis are presented in Section 4. In Section 5 we talk about related work, and draw conclusions in Section 6.

2. Unified Tasks and Conduits Framework

In this section we give an overview of the framework, describe how parallel applications are composed and then briefly show the framework interfaces for applications. Also we discuss the data sharing patterns and parallel programming methods that the framework provides.

2.1 Framework Overview

Based on the TNC model, we provide high level abstract components (Task/Conduit) for implementing parallel applications. In our framework, we leverage object oriented programming methods to express these two components. 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 basic objects.

In a task object, the data members contain the data that will be used for computation. The member functions define and implement specific computational logic or algorithms. The conduit object works as the bridge between different task objects, used to transfer data between task objects. To represent parallel program execution, we use a thread as the basic execution unit. Each task object is bound to one or several threads, and invokes threads to execute certain member functions. The overall UTC framework design is show as Figure 1.

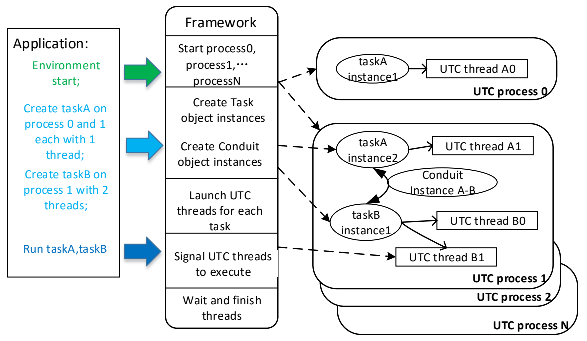

Application:
Environment start;
Create taskA on process 0 and 1 each with 1 thread;
Create taskB on process 1 with 2 threads;
Run taskA,taskB
Framework
Start process0, process1,··· processN
Create Task object instances
Create Conduit object instances
Launch UTC threads for each task
Signal UTC threads to execute
Wait and finish threads
taskA instance1
UTC thread A0
UTC process 0
taskA instance2
UTC thread A1
Conduit Instance A-B
taskB instance1
UTC thread B0
UTC thread B1
UTC process 1
UTC process 2
UTC process N

Figure 1: UTC framework overall design

In our design, we assign one process (`UTC_process`) to one compute node, thus each process represents a unique node in the cluster. One `UTC_process` stands for one compute node. Under this framework, a UTC-task is made up of a group of threads running on a single or multiple nodes. Threads of different tasks that run on the same node belong to the same `UTC_process`.

To start an application, N processes (`UTC_process`) are started by the system and execute the application program in a SPMD pattern. Every process has a unique id (from 0 to N-1). The subsequent procedures on one node happen in the corresponded `UTC_process`.

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 1.
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 1.

So in our framework, a task is made up of one or multiple `task object` instances and a group of threads which can perform computations in SPMD. Tasks can be dynamically created during the execution of programs. Different tasks run with different threads and 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.

With the help of this framework, we can define and create necessary tasks and conduits, 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 programming methods. In this way, the higher level logic of the application is isolated from lower level task implementation. When the platform changes, the application structure needs little or no modifications; we only focus on the adaptation of task implementation codes. For example, with the framework runtime support, we may realize tasks to be run on GPUs and replace the old CPU tasks with new versions of GPU tasks to make use of GPUs to accelerate the application.

2.2 Framework Interface

In the UTC framework, we provide a set of classes and utilities to define and run tasks in the system. A simple way to define and create a task is:

Task⟨`USER_TASK`⟩ task_instance(`proc_list`, ...);

There are two important required arguments:

1. `USER_TASK`: This is a user defined class template, based on which the `task object` is instantiated and bound to the execution threads. 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 necessary computational algorithms.
2. `proc_list`: This is a vector of `UTC_process` IDs. Each element of the vector indicates where threads will be launched and the size of the vector is the total number of threads 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;

A user just invokes these methods to inform all threads bound to the calling `task object` to perform the required actions. `init/wait/finish` are blocking methods. They will wait for the threads to finish a certain job. The `run` method is a nonblocking method and returns as soon as the command is submitted. So a user can invoke different tasks to run successively, no need to wait for an earlier tasks' completion.

We also implement the `Conduit` class, allowing the user to define conduit objects and pass them as arguments to tasks for data communication. Two basic conduit methods are: `Conduit::read` and `Conduit::write`, which are used for fetching and putting data through a conduit object.

2.3 Hybrid Data Sharing and Different Types of Parallelism

From the above description we learn that a task in the UTC framework is associated with a number of `task object` instances and a group of threads. In a task, threads from the same `UTC_process` are bound to the same `task object`. Therefore, for each task, there are three different data scopes:

local-private: data belongs to a singe thread, not shared.

local-shared: data are shared by threads on the same node, we call these threads `local_thread_group`.

global-shared: data are shared by all threads serving one task across multiple nodes.

In the UTC framework, local-shared is straightforward. Because threads on the same node are bound and share the same `task object`, all the ordinary data members of the object are shared by `local_thread_group`, and they are local-shared. To enable a user to create data members in a `task object` that can be local-private or global-shared, we define the following data types:

`PrivateScopedData`: Implement with thread-local storage (TLS) which is supplied by an underlying pthread or boost thread library. Each thread operates on the local copy of the data.

`GlobalScopedData`: Implement through underlying OpenSHMEM library [5], dynamically create OpenSHMEM symmetric memory space across multiple processes and utilize one-sided put/get operations to access data remotely on other nodes.

With the help of these data types, the UTC framework provides multiple levels and mixed data sharing mechanisms. It enables the user to use threads + PGAS hybrid programming methods to implement multi-threaded task programs. Through the global-shared data object, a task can run multiple threads on different nodes, not limited to a single node by the traditional thread programming method. The parallel threads of different nodes can communicate through global shared data with one-sided remote memory access (RMA) operations, which brings more convenience to a user to realize a parallel algorithm across multiple nodes. At the same time, unlike pure PGAS methods, multiple threads on the same node already share memory spaces, so there is no need to create global shared data for every parallel thread, reducing the memory pressure. In addition, using this hybrid threads + PGAS method, a problem space is divided and distributed by the number of nodes you are using. On each node, multiple threads share the sub-problem space and work on it. This distribution may reduce the inter-node communication in some applications.

Besides using hybrid threads + PGAS programming methods to implement a single task, a user can define and create several tasks dynamically in an application. These tasks runs 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 data parallelism. Between different UTC tasks, they are loosely coupled. They may collaborate through exchanging data using a conduit, or just run independently for task parallelism. A special case occurs when an application includes several consecutive parts. Then it is easy to create multiple tasks and implement a pipeline pattern.

Through the UTC framework, a user can implement coarse-grained MPMD parallelism with multiple tasks running concurrently; also a user can implement fine grained SPMD parallelism with multiple threads running in parallel for each task. Overall the UTC framework provides flexible and efficient means for the user to design and implement parallel applications for cluster platforms.

3. Framework Implementation and Discussion

3.1 Framework Implementation

The UTC framework runs on cluster platforms. It needs to start up processes locally and remotely as well as dynamically creating and managing threads within each process. It is able to exchange data between different tasks and share data among all threads of one task. Based on these requirements there are various tools and libraries that could be used for framework implementation.

3.1.1 Parallel Execution Creation

To start up multiple 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. Besides using MPI, it is also feasible to use other tools, like SLURM [16] or Active Message [15] to start up a process on a remote node.

To launch multiple threads in each process, we make use of Pthreads library to create and manage multiple threads. We leverage the native C++11 threads utilities which may require recent compiler support. Also we use some boost::thread functions for assistance.

3.1.2 Data Communication

In the framework, we use conduits for communication between different tasks. The conduit expresses the data dependencies. As two tasks may be active on the same node or on different nodes, we implement both intra-node conduits and inter-node conduits. For intra-node conduits, we use a shared memory method to build intermediate buffers for data movement. For inter-node conduits, we make use of MPI communication methods to transfer data between processes. More details about implementation are available [12].

Another important part of data communication is to support global data sharing within one task among multiple processes. Currently, we implement this feature through tools that support the PGAS model. In our runtime implementation we make use of the OpenSHMEM library [5] to accomplish our goal. We use OpenMPI as the MPI implementation in the system. OpenMPI also has realized the OpenSHMEM standard and provides the related methods.

Besides using MPI or OpenSHMEM as the network communication layer, we also could use network libraries such as GASNet [8] or UCCS [9] directly to build up the inter process data movement and sharing in the framework runtime system. In this way we may have more control and flexibility to implement the expected data communication behavior, and have better support for our multi-threaded environment. This also increase the difficulty to implement the framework runtime system.

3.2 Framework Extension for GPU task Support

Our framework design does not limit running tasks to multicore CPUs; it also supports creating and running tasks on accelerators. Programs that make use of accelerators for execution are not executed on accelerators alone. There must be control flow (host-control) that runs on CPUs, and computing flow (device-kernel) that runs on the accelerator. The host-control part is the same as a normal thread which runs on CPU. To integrate the device-kernel part, we have defined a set of interfaces for extension, such as `acc_init`, `acc_device_bind`, `acc_kernel_launch` and so on. To target a specific accelerator, we need to implement these interfaces and some glue code in the runtime system to be able to manage and control GPUs. From the application perspective, the "accelerator task" behaves the same as a "CPU task", except that it launches a kernel on an accelerator for computing.

Task⟨`USER_TASK`⟩ ctask_instance(`proc_list`, **"cpu_task"**, ...);
Task⟨`USER_TASK`⟩ gtask_instance(`proc_list`, **"gpu_task"**, ...);

Targeting Nvidia GPUs and leveraging CUDA runtime and methods, we have implemented a preliminary GPU tasks extension in our framework runtime system. A "gpu task" is also associated with multiple threads, but each thread will be bound to a GPU device, executing the host-control workloads and invoking the CUDA kernels to launch massive CUDA threads on GPUs for computing. There is no difference for a user to define and use a "gpu task", however, the task's computational kernel must be implemented through CUDA or another programming method.

4. Application Tests

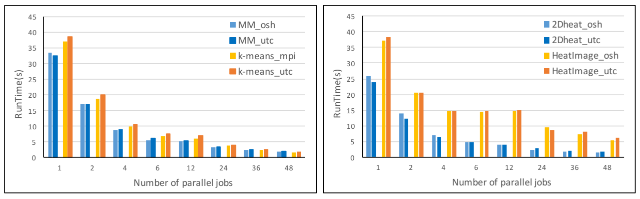
We developed several applications to test and evaluate our framework implementation. One experimental platform is a small cluster comprised of four computing nodes, connected by Gigabit Ethernet. Each node has an Intel Xeon E-2620 CPU (6 cores with hyper-threading support). Another platform is a desktop server equipped with one Nvidia C2070 GPU. There are four applications used for testing. Three of them: Matrix Multiply (MM), 2D Heat Conduction Simulation (2Dheat) and Heat Image generation (HeatImage) are ported based on the test suite example programs from the OpenSHMEM (OSH) website [6]. The k-means program is a widely used clustering application. Some input data set information is shown in Table 1.

Name	Data set information
Matrix Multiply	Matrix size: 2880*2880(double).
2D Heat	2D plain size: 288*800(single)
Heat Image	Image size: 2880*1000(double)
k-means	1600*1066 points, 5 attributes(single)

Table 1: Test Applications Info

4.1 Multicore CPUs Execution Test

We first run these test programs on the cluster platform, scaling up and using various number of parallel jobs for execution. MM and 2Dheat both have only one primary core computation procedure, so there is one task in each program. In HeatImage and k-means programs, there are two procedures: one for computation and another for file read or write. So we implement them as two task templates.



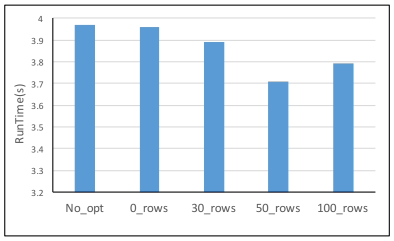
(a) (b)

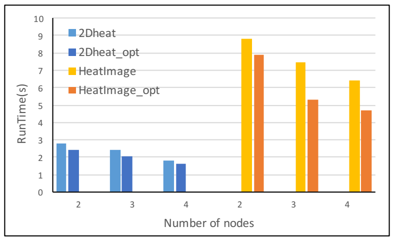
Figure 2: Application run-time tests: (a) MM and k-means;(b) 2Dheat and HeatImage.(Up to 12 parallel jobs on one node)

The applications run-time test results are show in Figure 2. In our test platform each node has 6 cores with 2 hyper-threads, thus 4 nodes support 48 parallel threads or processes. In the tests we run at most 12 parallel jobs on one node, so when the number of jobs is more than 12 we use multiple nodes for execution. From the test results we can see that when increasing parallel jobs from 6 to 12, the run-time does not change much, indicating that hyper-threading does not bring much parallel execution ability for these applications. When running with more nodes, the run-time improvements are not as much as on a single node. This is because the inter-node communication and synchronization latency is higher than on a single node; also the workload of each job is already low which makes the effect of communication and synchronization more apparent. Generally, we can see that programs implemented with our framework have comparable performance to traditional parallel programming methods. But with the use of high level task and conduit primitives for programming, the UTC programs have a clear and concise structure, which brings better portability and maintenance.

4.2 Multi-threaded Task Optimization

In our framework, we design the parallel algorithm for each task using threads + PGAS programming method. Multiple threads on the same node can easily share data, and threads of different nodes can share data through global shared data objects. The explicit multiple threads of a task gives the user more flexibility to manipulate





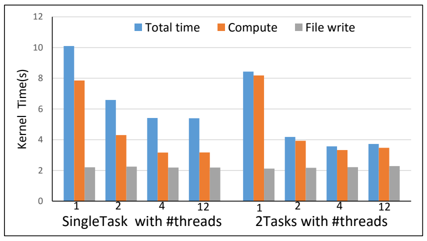


Figure 3: MM overlapping test on two nodes Figure 4: 2Dheat and HeatImage overlapping test on multiple nodes

Figure 5: Pipeline execution test

and use parallel threads for program optimization. One optimization approach is overlapping computation and communication with multiple threads.

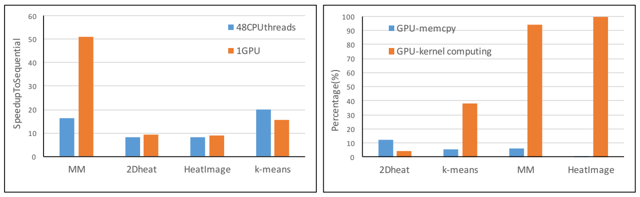
In the Matrix Multiply application ($A \times B = C$), we implement a block-column distribution for all three matrices. Each parallel job holds one column block of A, B and C. When running multiple nodes, it is easy for us to schedule one thread (comm-thread) to fetch data from the neighbor nodes for future use while the other threads keep busy with computing (comp-thread). In addition to using one thread for communication, we can add small computational workloads to the comm-thread to keep it busier. We also test this scenario. Figure 3 shows the test results of running MM on two nodes. In Figure 3, "No_opt" is the run-time without overlapping optimization. "0_rows" means we do not assign additional computation to comm-thread, while "30_rows", "50_Rows" and "100_Rows" means we let comm-thread compute 30, 50 or 100 rows of points in addition to do the communication. We can see that without adding extra computational workload, we do not get run-time improvement. But adding too much extra computation to the comm-thread, results in a degradation of performance because of the workload imbalance of different threads. We also adopt the same optimization on 2Dheat and HeatImage programs and figure 4 shows the test results.

4.3 Pipeline Execution Test

In the Heat Image Generation example, there are two procedures: one is to compute the heat image data set; the other is to write the heat image data to a file on disk. We made some modifications to the program: we select and save one image to file for every 100 iterations. 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 image data to a conduit; the other one reads data from the conduit and saves it to files. As both tasks are running with different threads, the file writing task can run concurrently with the data computation task. The UTC framework enables them to run in a pipelined pattern easily. Test results are shown in Figure 5. 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 for 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.

4.4 GPU Execution Test

In addition to implementing and running tasks on multicore CPU cluster platforms, we also implement the GPU tasks for the above applications. The GPU we used is a Nvidia Tesla C2070 GPU, which has 448 CUDA cores and 6GB on device memory. We implement the necessary GPU tasks for each application with CUDA and use these GPU task implementations as the class template to instantiate tasks. The main structure of the original UTC programs is unchanged. In the tests, we only realize the basic CUDA implementations for each algorithm and do not adopt any sophisticated optimizations. We compare the performance of programs run on a single GPU to those executed on 4 CPU nodes with 48 threads. The results are shown in Figure 6a. Here speedup refers to a comparison with single threaded, sequential code.



(a) (b)

Figure 6: GPU task tests: (a) GPU task speedup;(b)Percentage of different work in total run-time.

We can see that GPU task performance depends on the application. For MM, a single GPU performs much better than 4 CPU nodes. But the other three programs' performances of GPU and CPU are similar. One reason may be that our GPU implementations are not specially tuned. In some applications, such as 2Dheat and k-means, we need to do some reduction-like computation to check the convergence state between every iteration. This work is not implemented on the GPU and becomes a bottleneck for overall performance. Also, when porting tasks to GPU, the memory copy procedures of transmitting data between main memory and GPU memory may influence the performance. In Figure 6b we show the run-time percentages of memory copy and GPU kernel in the total program run-time. We can see that the GPU kernel runtime percentage in both k-means and 2Dheat is not very high, so the workload on the CPU limits the overall performance. Also in 2Dheat test, we can see that the time cost of data transfer between GPU memory and main memory is even more than the actually time spent on computation on GPU, so the memory copy becomes the bottleneck for this application. However, here we only run with one GPU. If running on GPU clusters with multiple GPUs or running with more powerful newer GPUs, we will achieve even more performance improvement compared to only running with CPU tasks. By implementing appropriate GPU tasks with CUDA and replacing the task templates in the main program, we can easily port the program to GPU platforms to leverage GPUs for acceleration in our framework.

5. Related Work

As multicore and manycore processors become prevalent, there is growing research interest in parallel programming and developing parallel applications to benefit from changing platforms. Traditional parallel programming models, such as OpenMP and MPI, are still the most widely used and there are lots of legacy applications based on them. OpenMP helps users run code blocks in parallel with multiple threads on shared memory systems, and is especially suitable for dealing with data parallelism in loop structures. OpenMP3.0 introduces "#pragma task" to better support task parallelism, and OpenMP4.0 [13] introduces "#pragma target" to support offloading code to accelerators. This ability relies on compiler support, such as Intel OpenMP for Intel Xeon Phi [11]. In the UTC framework, we create threads based on system thread libraries explicitly. This gives the user more control and flexibility to manipulate the threads. In addition, our framework is compatible with OpenMP. It is able to make use of OpenMP for deeper fine grained data parallelism in task execution.

The PGAS model introduces a distributed shared memory space to ease programming effort with a shared memory model and one-sided communications. There are a series of languages and libraries belonging to the PGAS family, including Unified Parallel C [10], SHMEM [7], OpenSHMEM [5]. By extending the PGAS model with dynamic asynchronous activities, Asynchronous PGAS (APGAS) [17] is proposed. X10 [14] is a representative programming language of APGAS. Through extending the Java language, X10 introduces *place* and *async* keywords for the user to define and create parallel activities. The runtime system will convert a function declared with *async* to a task (an activity) that will be scheduled to run asynchronously. Each *async* functions's input/output information must be provided by the user and the X10 runtime will use this info to build a Directed Acyclic Graph (DAG). Each function is then scheduled to run based on this graph. Besides X10, other APGAS programming languages such as ClusterSs [18], OmpSs [19] are also introduced. These programming methods try to help the user express task parallelism in an application more easily, and make use of parallel resources to improve application performance. However, the tasks created in these methods are usually very lightweight sequential programs, such as small functions or several lines of statements in a loop structure. Also, they follow and extend the Java language and use a front-end compiler to preprocess the programs. This makes it difficult to support new processing devices such as GPUs or FPGAs. We use the C++ language and implement our framework through a library-based approach, which is easier to extend to integrate new hardware, such as running tasks on GPUs.

6. Conclusions

In this paper we propose a parallel programming framework: UTC. In the framework, we leverage the TNC model to describe task parallelism explicitly for applications. Users can define and create multiple tasks to run on a single node or multiple nodes easily and naturally. For a single task, we leverage multiple threads and a global shared data object to create a distributed shared memory environment, allowing users to design and implement data parallelism through hybrid threads + PGAS programming. Through the UTC framework, users can explore both data parallelism and task parallelism for an application efficiently, as well as separating the high level application structure from low level task implementation and execution. By supporting GPU tasks, our framework runtime allows users to develop and run parallel applications on heterogeneous platforms. Test results show that our framework runtime introduces little performance overhead to applications. It is easy to apply pipelined execution or overlap computation with computation to improve parallel applications' performance. Without changing the program main structure, we also implement the GPU version of required tasks and make use of GPUs for parallel execution. The results show that the performance of a single GPU is better than four CPU computer nodes. In the future we plan to improve the framework's runtime implementation, especially the global shared data implementation. Our current GPU task support is still in a preliminary stage; there are several aspects for future exploration, such as the use of multiple GPUs, and GPU memory integration.

References

[1] K. Yelick, D. Bonachea, W.-Y. Chen, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.

[2] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[3] Nvidia. CUDA C programming guid. URL `http://docs.nvidia.com/cuda/pdf/`.

[4] 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.

[5] 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*, page 2. ACM, 2010.

[6] OpenSHMEM test suites. URL `https://github.com/openshmem-org/test-suites`.

[7] R. Barriuso and A. Knies. Shmem user's guide for C. Technical report, Technical report, Cray Research Inc, 1994.

[8] D. Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[9] A. Bouteiller, T. Herault, and G. Bosilca. A multithreaded communication substrate for openshmem. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 16. ACM, 2014.

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

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

[12] C. Liu and M. Leeser. Unified and lightweight tasks and conduits: A high level parallel programming framework. In *IEEE High Performance Extreme Computing Conference*, 2016.

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

[14] 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*, pages 271–271. ACM, 2007.

[15] 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*, volume 20, pages 256–266. ACM, 1992.

[16] 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*, pages 44–60. Springer, 2003.

[17] 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.

[18] 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.

[19] 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.