## Introduction to MPI-2

For a continuous development of MPI, the MPI Forum has defined extensions to MPI. These extensions are often referred to as MPI-2. The original MPI standard is referred to as MPI-1. Since MPI-2 comprises all MPI-1 operations, each correct MPI-1 program is also a correct MPI-2 program. The most important extensions contained in MPI-2 are dynamic process manage-ment, one-sided communications, parallel I/O, and extended collective communica-tions. In the following, we give a short overview of the most important extensions.

## Dynamic Process Generation and Management

MPI-1 is based on a **static process model**: The processes used for the execution of a parallel program are implicitly created before starting the program. No processes can be added during program execution. Inspired by PVM [Parallel Virtual Machine], MPI-2 extends this process model to a **dynamic process model** which allows the creation and deletion of processes at any time during program execution. MPI-2 defines the interface for dynamic process management as a collection of suitable functions and gives some advice for an implementation. But not all implementation details are fixed to support an implementation for different operating systems.

## MPI Info Objects

Many MPI-2 functions use an additional argument of type MPI Info which allows the provision of additional information for the function, depending on the spe-cific operating system used. But using this feature may lead to non-portable MPI programs. MPI Info provides opaque objects where each object can store arbi-trary (key, value) pairs. In C, both entries are strings of type char, terminated with \0. Since MPI Info objects are opaque, their implementation is hidden from the user. Instead, some functions are provided for access and manipulation. The most important ones are described in the following. The function

```
int MPI_Info_create (MPI_Info *info)
```

can be used to generate a new object of type MPI Info.

Calling the function

```
int MPI_Info_set (MPI_Info info, char *key, char *value)
```

adds a new (key, value) pair to the MPI Info structure info. If a value for the same key was previously stored, the old value is overwritten. The function

```
int MPI_Info_get (MPI_Info info,
                  char *key,
                  int valuelen,
                  char *value,
                  int *flag)
```

can be used to retrieve a stored pair (key, value) from info. The programmer specifies the value of key and the maximum length valuelen of the value entry. If the specified key exists in info, the associated value is returned in parameter value. If the associated value string is longer than valuelen, the returned string is truncated after valuelen characters. If the specified key exists in info, true is returned in parameter flag; otherwise, false is returned. The function

```
int MPI_Info_delete(MPI_Info info, char *key)
```

can be used to delete an entry (key, value) from info. Only the key has to be specified.

## Process Creation and Management

A number of MPI processes can be started by calling the function

```
int MPI_Comm_spawn (char *command,
                    char *argv[],
                    int maxprocs,
                    MPI_Info info,
                    int root,
                    MPI_Comm comm,
                    MPI_Comm *intercomm,
                    int errcodes[]).
```

The parameter command specifies the name of the program to be executed by each of the processes, argv[] contains the arguments for this program. In contrast to the standard C convention, argv[0] is not the program name but the first argument for the program. An empty argument list is specified by MPI ARGV NULL. The param-eter maxprocs specifies the number of processes to be started. If the MPI runtime system is not able to start maxprocs processes, an error message is generated. The parameter info specifies an MPI Info data structure with (key, value) pairs providing additional instructions for the MPI runtime system on how to start the processes. This parameter could be used to specify the path of the program file as well as its arguments, but this may lead to non-portable programs. Portable programs should use MPI INFO NULL.

The parameter root specifies the number of the root process from which the new processes are spawned. Only this root process provides values for the preced-ing parameters. But the function MPI Comm spawn() is a collective operation, i.e., all processes belonging to the group of the communicator comm must call the function. The parameter intercomm contains an intercommunicator after the suc-cessful termination of the function call. This intercommunicator can be used for communication between the original group of comm and the group of processes just spawned.

The parameter errcodes is an array with maxprocs entries in which the status of each process to be spawned is reported. When a process could be spawned successfully, its corresponding entry in errcodes will be set to MPI SUCCESS. Otherwise, an implementation-specific error code will be reported.

A successful call of MPI Comm spawn() starts maxprocs identical copies of the specified program and creates an intercommunicator, which is provided to all calling processes. The new processes belong to a separate group and have a separate MPI COMM WORLD communicator comprising all processes spawned. The spawned processes can access the intercommunicator created by MPI Comm spawn() by calling the function

```
int MPI Comm get parent(MPI Comm *parent).
```

The requested intercommunicator is returned in parameter parent. Multiple MPI programs or MPI programs with different argument values can be spawned by call-ing the function

```
int MPI Comm spawn multiple (int count,
                            char *commands[],
                            char **argv[],
                            int maxprocs[],
                            MPI Info infos[],
                            int root,
                            MPI Comm comm,
                            MPI Comm *intercomm,
                            int errcodes[])
```

where count specifies the number of different programs to be started. Each of the following four arguments specifies an array with count entries where each entry has the same type and meaning as the corresponding parameters for MPI Comm spawn(): The argument commands[] specifies the names of the programs to be started, argv[] contains the corresponding arguments, maxprocs[] defines the number of copies to be started for each program, and infos[] provides additional instructions for each program. The other arguments have the same meaning as for MPI comm spawn().

After the call of MPI Comm spawn multiple() has been terminated, the array errcodes[] contains an error status entry for each process created. The entries are arranged in the order given by the commands[] array. In total, errcodes[] contains

count-1

maxprocs[i]

i=0

entries. There is a difference between calling MPI Comm spawn() multiple times and calling MPI Comm spawn multiple() with the same arguments. Calling the function MPI Comm spawn multiple() creates one communicator MPI COMM WORLD for all newly created processes. Multiple calls of MPI Comm spawn() generate separate communicators MPI COMM WORLD, one for each pro-cess group created.

The attribute MPI UNIVERSE SIZE specifies the maximum number of pro-cesses that can be started in total for a given application program. The attribute is initialized by MPI Init().

## Accelerators

Accelerators are dramatically more efficient than CPUs for their target applications. However, they are not as flexible and perform poorly on other workloads.

GPUs are programmable computational accelerators that aim to accelerate a wide range of parallel applications. Early GPUs were designed as high compute density, fixed function processors ideally crafted to the needs of computer graphics workloads. Over time, however, they gained increasingly general purpose capabilities.

**OpenACC** (for *open accelerators*) is a programming standard for parallel computing developed by Cray,CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems. As in OpenMP, the programmer can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using compiler directives and additional functions. OpenACC can target both the CPU and GPU architectures and launch computational code on them.

Thus, OpenACC Application Programming Interface (API) provides a set of compiler directives, library routines, and environment variables that can be used to write data-parallel FORTRAN, C, etc. programs that run on accelerator devices, including GPUs.

## CUDA

CUDA is a parallel computing platform and an API model that was developed by Nvidia. Using CUDA, one can utilize the power of Nvidia GPUs to perform general computing tasks, such as multiplying matrices and performing other linear algebra operations, instead of just doing graphical calculations. Using CUDA, developers can now harness the potential of the GPU for general purpose computing (GPGPU). CUDA − **C**ompute **U**nified **D**evice **A**rchitecture is an extension of C programming, an API model for parallel computing. Programs written using CUDA harness the power of GPU. Thus, increasing the computing performance.

## Key Concepts

### Data Parallelism

Modern applications process large amounts of data that incur significant execution time on sequential computers. An example of such an application is rendering pixels. For example, an application that converts sRGB pixels to grayscale. To process a 1920x1080 image, the application has to process 2073600 pixels.

Processing all those pixels on a traditional uniprocessor CPU will take a very long time since the execution will be done sequentially. (The time taken will be proportional to the number of pixels in the image). Further, it is very inefficient since the operation that is performed on each pixel is the same, but different on the data (SPMD). Since processing one pixel is independent of the processing of any other pixel, all the pixels can be processed in parallel. If we use 2073600 threads (“workers”) and each thread processes one pixel, the task can be reduced to constant time. Millions of such threads can be launched on modern GPUs.

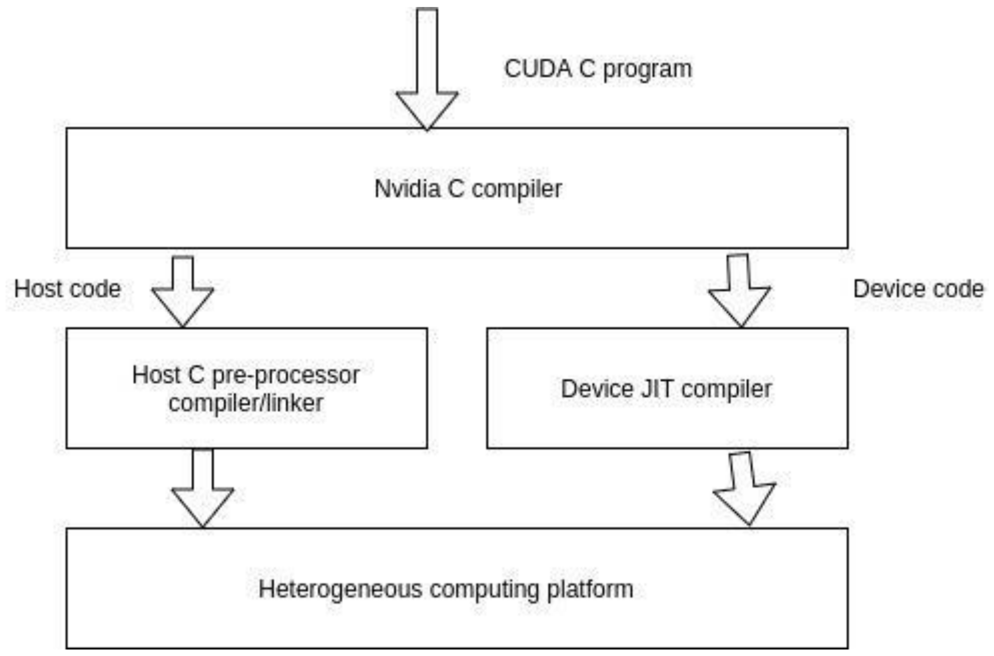
GPU is traditionally used for rendering graphics. For example, per-pixel lighting is a highly parallel, and data-intensive task and a GPU is perfect for the job. We can map each pixel with a thread and they all can be processed in O(1) constant time.

Image processing and computer graphics are not the only areas in which we harness data parallelism to our advantage. Many high-performance algebra libraries today such as CU-BLAS harness the processing power of the modern GPU to perform data intensive algebra operations. One such operation is matrix multiplication.

### Program Structure of CUDA

A typical CUDA program has code intended both for the GPU and the CPU. By default, a traditional C program is a CUDA program with only the host code. The CPU is referred to as the host, and the GPU is referred to as the device. Whereas the host code can be compiled by a traditional C compiler as the GCC, the device code needs a special compiler to understand the api functions that are used. For Nvidia GPUs, the compiler is called the NVCC (Nvidia C Compiler).

The device code runs on the GPU, and the host code runs on the CPU. The NVCC processes a CUDA program, and separates the host code from the device code. To accomplish this, special CUDA keywords are looked for. The code intended to run of the GPU (device code) is marked with special CUDA keywords for labelling data-parallel functions, called ‘Kernels’. The device code is further compiled by the NVCC and executed on the GPU.



## Execution of a CUDA C Program

While writing a CUDA program, the programmer has explicit control on the number of threads that he wants to launch (this is a carefully decided-upon number). These threads collectively form a three-dimensional grid (threads are packed into blocks, and blocks are packed into grids). Each thread is given a unique identifier, which can be used to identify what data it is to be acted upon.

Device Global Memory and Data Transfer

A typical GPU comes with its own global memory (DRAM- Dynamic Random Access Memory). For example, the Nvidia GTX 480 has DRAM size equal to 4G known as the device memory.

To execute a kernel on the GPU, the programmer needs to allocate separate memory on the GPU by writing code. The CUDA API provides specific functions for accomplishing this. Here is the flow sequence −

- After allocating memory on the device, data has to be transferred from the host memory to the device memory.
- After the kernel is executed on the device, the result has to be transferred back from the device memory to the host memory.
- The allocated memory on the device has to be freed-up. The host can access the device memory and transfer data to and from it, but not the other way round.

CUDA provides API functions to accomplish all these steps.

## OpenCL

OpenCL is a standardized, cross-platform parallel computing API based on the C language. It is designed to enable the development of portable parallel applications for systems with heterogeneous computing devices. The development of OpenCL was motivated by the need for a standardized high-performance application development platform for the fast-growing variety of parallel computing platforms. In particular, it addresses significant application portability limitations of the previous pro- gramming models for heterogeneous parallel computing systems.

CPU-based parallel programming models have been typically based on standards such as OpenMP but usually do not encompass the use of special memory types or SIMD (single instruction, multiple data) execution by high-performance programmers. Joint CPUGPU heterogeneous parallel programming models such as CUDA have constructs that address complex memory hierarchies and SIMD execution but have been platform-, vendor-, or hardware-specific. These limitations make it difficult for an application developer to access the computing power of CPUs, GPUs, and other types of processing units from a single multiplatform source code base.

The development of OpenCL was initiated by Apple and managed by the Khronos Group, the same group that manages the OpenGL standard. On one hand, it draws heavily on CUDA in the areas of supporting a sin- gle code base for heterogeneous parallel computing, data parallelism, and complex memory hierarchies.

On the other hand, OpenCL has a more complex platform and device management model that reflects its support for multiplatform and multi- vendor portability. OpenCL implementations already exist on AMD/ATI and NVIDIA GPUs as well as X86 CPUs. In principle, one can envision OpenCL implementations on other types of devices such as digital signal processors (DSPs) and field programmable gate arrays (FPGAs). While the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come for free. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. Also, many OpenCL fea- tures are optional and may not be supported on all devices. A portable OpenCL code will need to avoid using these optional features. However, some of these optional features allow applications to achieve significantly more performance in devices that support them. As a result, a portable OpenCL code may not be able to achieve its performance poten- tial on any of the devices. Therefore, one should expect that a portable application that achieves high performance on multiple devices will employ sophisticated runtime tests and chose among multiple code paths according to the capabilities of the actual device used.

## PGAS

Threading and message passing are the two dominant programming models on current parallel systems. Programming with threads on a shared memory system is conceptually simpler than coordinating the explicit sending and receiving of messages. However, the lack of control over data locality can hamper performance and true shared memory systems are limited in their size to a few dozen cores.

The Partitioned Global Address Space (PGAS) approach is a promising programming model in high performance parallel computing that combines the advantages of distributed memory systems and shared memory systems. The PGAS model has been used on a variety of hardware platforms in the form of PGAS programming languages like Unified Parallel C (UPC), Chapel and Fortress.

PGAS (Partitioned Global Address Space) approaches try to bring the advantages of shared memory style programming to large scale distributed systems. A PGAS language or library uses put and get operations to local and remote memory locations to provide a programming model that is very similar to programming with threads. Additionally, the locality (affinity to processing elements) of data is made explicit to enable efficient program development. The efforts of realizing PGAS models have concentrated primarily on multi-core and multi-chip platforms based on message-passing and shared memory techniques , while accelerator architectures have rarely been focused on. With the widespread adoption of GPGPU (General Purpose Computingon GPUs) and generally accepted GPU programming models like CUDA and OpenCL gaining popularity, possibilities and demands for extending and unifying CPU memory spaces with GPU memory spaces in GPU-equipped systems arise in the field of high performance computing.

## Distributed Memory Architectures

This architecture enables scalability by distributing the memory throughout the machine, using a scalable interconnect to enable processors to communicate with the memory modules. Based on the communication mech- anism provided, these architectures are classified as:

• Multicomputer/message-passing architectures
• DSM architectures

The multicomputers use a software (message-passing) layer to communi- cate among themselves and hence are called message-passing architectures. In these systems, programmers are required explicitly to send messages to request/send remote data. As these systems connect multiple computing nodes, sharing only the scalable interconnect, they are also referred to as multicomputers. DSM machines logically implement a single global address space although the memory is physically distributed. The memory access times in these systems depended on the physical location of the processors and are no longer uniform. As a result, these systems are also termed nonuniform memory access (NUMA) systems.

### CLASSIFICATION OF DISTRIBUTED SHARED MEMORYSYSTEMS

Providing DSM functionality on physically distributed memory requires the implementation of three basic mechanisms:

Processor-side hit/miss check. This operation, on the processor side, is used to determine whether or not a particular data request is satisfied in the processor’s local cache. A hit is a data request satisfied in the local cache; a miss requires the data to be fetched from main memory or the cache of another processor.

• Processor-side request send. This operation is used on the processor side in response to a miss, to send a request to another processor or main memory for the latest copy of the relevant data item and waits for even- tual response.

• Memory-side operations. These operations enable the memory to receive a request from a processor, perform any necessary coherence actions, and send its response, typically in the form of the data requested.

Depending on how these mechanisms are implemented in hardware or soft- ware helps classify the various DSM systems as follows:

• Hardware-based DSM systems. In these systems, all processor-side mech- anisms are implemented in hardware, while some part of memory-side support may be handled in software. Hardware-based DSM systems include SGI Origin ,HP/Convex Exemplar ,MIT Alewife ,and Stanford FLASH .

• Mostly software page-based DSM systems. These DSM systems implement hit/miss check in hardware by making use of virtual memory protection mechanisms to provide access control. All other support is implemented in software. Coherence units in such systems are the size of virtual memory pages. Mostly software page-based DSM systems include TreadMarks , Brazos , and Mirage+ .

• Software/Object-based DSM systems. In this class of DSM systems, all three mechanisms mentioned above are implemented entirely in soft- ware. Software/object-based DSM systems include Orca , SAM , CRL , Midway , and Shasta.