Proceedings of the International Workshop on Intelligent Information Systems
IIS'2011, September 13-14, 2011, Chisinau, Republic of Moldova

Architecting a Hybrid Computation System Using Microsoft Parallel Extensions, Open MPI, nVidia CUDA

Veaceslav Macari

Abstract: The article describes the possibility to develop a framework which uses hybrid parallel computing to achieve maximal performance from available hardware resources (CPU & GPU) of all computers in LAN.
Keywords: hybrid parallel computing, CUDA, Open MPI, Microsoft Parallel Extensions

1 Introduction

There are already well known programming technologies used in modern world: MS Parallel Extensions (MS PE), MPI, and CUDA. Hybrid programming supposes usage of several parallel programming techniques together to achieve maximal performance of algorithms.

The idea of proposed framework is to use combination of the mentioned technologies and provide a set of APIs that will hide underlaying hardware from users. This implies creation of a virtual machine that offers a layer between existing hardware and user application, to provide abstract calculation resources. It should be developed as independent functionality (using factory pattern), flexible and runable even it is executed on a single CPU machine, without network and GPU support.

The general architecture of a hybrid parallel system is composed from a group of cells, which uses MPI to distribute tasks at the network level. Then, at the entity (cell) level, MS PE is used to distribute processing task between each CPU. At the same time, at the cell level, CUDA is used to distribute tasks across available GPU's.

2 Used programming technologies description

MS Parallel Extensions is a managed concurrency library that implements multi-threading [1]. MS PE spreads tasks over available CPUs. There is always a master thread, which starts all other threads with unique id's that run concurrently on different processors. MS PE tasks run on the same host, but on different hardware processors.

MS PE in this case, is used to involve all CPUs present on the host computer, because usually, even if the user implements multiple threads with *fork ()* function, the tasks created by the same application share a single CPU. For an optimal usage of MS PE in application, the algorithms should be organized to allow division on parallel computing parts.

Open MPI - Open Message Passing Interface allows processes to communicate with one another by sending messages even if these processes are running on different computers on a high speed network.

Currently, Open MPI is implemented in different languages, including C/C++ and C# [2].

CUDA - Compute Unified Device Architecture is a parallel computation architecture elaborated by nVIDIA to be used with their GPU (Graphics Processing Units) devices [3]. The technology is ported to different languages like C, C++, Java and C# [4] because it uses so called kernel modules which are uploaded to CUDA enabled GPU, and serves as micro-routines that process data in concordance with algorithm requirements [5].

The kernels can be implemented using modular programming model, as separate CUDA object files, which are loaded at runtime and supplied to the GPU. Therefore algorithmic part is separated from management part and can be independently maintained without recompiling entire application.

3 Proposed framework architecture

The technologies **MS PE, Open MPI, and CUDA** are used together to involve all available resources to resolve a given task. The resources are allocated in the order of their performance decreasing: the most productive resource is used firstly.

For example, let's say application is requested to create 2 processes which perform computation of a huge matrix, and 2 other ones which execute common algorithms. The local configuration is a PC with a single CPU and single CUDA enabled GPU. In this case, the virtualization layer (VL) allocates its own GPU for one of the matrix computation algorithms and its own CPU for one of the common algorithms. Since the resources are already used, the VL searches for suitable node from its resource map and allocates the requested resources from a node of local area network.

The main role of the **virtualization layer** is to recognize algorithms and to distribute them between machines in a high speed LAN, on

processors of a single node for common operations, and to route to available GPU's of the same node for scalar operations.

The layer should provide an abstraction in such a way, that the programming model issues calculation request without taking care of underlaying hardware and processing resources. The number and performance of available resources affect only the performance of designed algorithms. The functions provided by the framework look like this: *virtualLayer.Matrix.Multiply(m1, m2)*, or if addressing to multitasking which is realized by MS PE: *virtualLayer.Threading.ExecuteParallel(() => Add(2, 3))*.

Such requests stimulate VL to search for a best suitable resource (see Table 1) and allocate it to execute requested operation.

Virtualization layer uses a map of available resources and resources topology for optimal allocation when they are requested by the user application. This map is built on layer initialization and is continuously updated during its life time period. Therefore new entries can be added, or the expired ones can be removed.

Table1: Resources map example

HOST_ID	CPU_CNT	GPU_NR	REACH_ COEF	PERFORMANCE	EXPIRITY_ TIMEOUT
Resource1	6	2	2	100	500
Resource2	9	1	5	400	700

The map includes the following information:

- **HOST_ID -** the IP/network address of available node;
- **CPU_CNT -** number of available CPUs;
- **GPU_NR -** number of available GPU;
- **REACH_COEF** – reachability coefficient, calculated from ping time and response delay;
- **PERFORMANCE -** calculated resource performance index, used to determine processing request across the involved nodes;
- **EXPIRITY_TIMEOUT** - permanently updated value, used to exclude offline hosts.

The distribution of data processing (see Fig.1) should be done across available processing resources - on demand. The calculation is launched from a master computer, which initiates the entire infrastructure as follows:

• Initiate and detect local processing capabilities (MP+CUDA);

• Discover similar hosts in the local network (MPI);

• Query the discovered hosts for their own processing capabilities (MP+CUDA);

• Compose an internal map of available resources, including performance and resources reachability (ping delay).

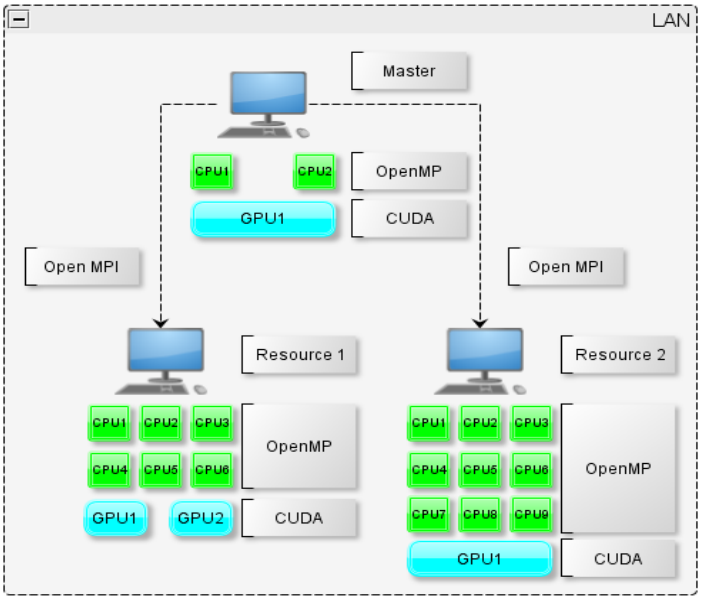

Figure 1. An architecture overview with data processing distribution using MS PE, Open MPI, and CUDA.

References

[1] http://en.wikipedia.org/wiki/Parallel_Extensions
[2] http://osl.iu.edu/research/mpi.net/
[3] http://developer.nvidia.com/category/zone/cuda-zone
[4] http://www.hoopoe-cloud.com/Solutions/CUDA.NET/Default.aspx#releases
[5] http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf , CHAPTER 2, page 5.

V. Macari

Institute of Mathematics and Computer Science of ASM
E-mail: vmacari@gmail.com