HPXNN

AMRITESH, PES University
ASHWATH KRISHNAN, PES University
SRIKAR S, PES University

HPX, short for High Performance ParalleX, is a runtime system for high performance computing. The project aims at implementing HPX in the C++ backends of various machine learning libraries with the help of a converter tool that has been built to achieve considerable performance improvements. The machine learning libraries used in the project to test HPX are Pytorch and OpenNN. OpenNN is a software library written in C++, that is used for the implementation of neural networks in the area of deep learning research. The project utilises the HPX runtime system to run individual tasks on the system cores to achieve better execution speeds.

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Parallel System, HPX, Open-Source, PyTorch, Machine Learning, OpenNN

ACM Reference Format:
Amritesh, Ashwath Krishnan and Srikar S, 2020. HPX implemented in Machine Learning libraries like PyTorch and OpenNN. *1st CAPS-PSRL Symposium of Summer of Systems* 9, 4, Article 39 (March 2010), 13 pages.
DOI: 0000001.0000001

1. INTRODUCTION

HPX, short for High Performance ParalleX, is a C++ runtime system for high performance computing. It represents an innovative mixture of a global system-wide address space (**AGAS** - Active Global Address Space), fine grain parallelism combined with an implicit, work queue based, message driven computation. HPX offers a dependable "futurized" API and has the following objectives:

— To promote distributed computing by executing tasks on a multi-node machine.
— To achieve concurrency by enabling multiple tasks to work on shared resources.
— To support parallelism, where several tasks work on the same core algorithm.

In today's world, there is a need for a highly adaptive runtime system that performs and scales well. 'Scalability' is the property of a system to handle a growing amount of work by adding resources to the system. The main factors that prevent scaling are the SLOW factors: (a) **S**tarvation, that occurs when the current running task is unable to utilize all the available resources, (b) **L**atency, that refers to the delay in accessing remote resources, (c) **O**verhead, which refers to the work done for the management of parallel actions, and (d) **W**aiting for Contention resolution, which is caused by delays due to shared resources. To overcome the above mentioned problems, a modern exe-

This work is supported by Centre for Advanced Parallel Systems - Parallel Systems Research Lab (CAPS-PSRL) PES University, 100 Feet Ring Road, BSK III Stage, Bangalore-560085
Author's addresses: Amritesh (mailto:amriteshc101@icloud.com), Ashwath Krishnan (mailto: ashwathkrishnan45@gmail.com), Srikar S (mailto:srikars2001.official@gmail.com), Department of Computer Science, PES University, 100 Feet Ring Road, BSK III Stage, Bangalore-560085

cution model is required and HPX has been designed to implement such an execution model. This is achieved by the use of “Lightweight Threads”.

In the CPU, it is the job of the operating system to manage ’hardware level’ or ’kernel level’ OS threads. But OS threads are expensive to create and destroy and take a time slice of CPU time. Moreover, the presence of multiple OS threads can degrade performance. Hence, HPX utilises lightweight threads, that can share address space and resources with other threads, reducing context switching time during execution.This allows for lightweight task scheduling and also optimizes performance. Therefore, on startup, HPX creates one ’worker’ thread per core and on each worker thread, HPX runs its own Task Scheduler. HPX tasks are then executed on the HPX worker thread, where each task is referred to as a lightweight thread. With HPX, the runtime is started on program setup and stays active until program termination. All tasks run on a HPX thread and the runtime can be manually started/stopped.

HPX aims to provide high quality, free open source parallelizing techniques. It has a well defined and modern runtime system. Moreover, it enables programmers to write asynchronous code and use millions of threads. It provides runtime adaptivity and supports the standard C++ API for ease of parallel programming and distributed applications. It is equipped with a unified system of semantics and helps in local and remote operations. Moreover, with future based synchronization, it manages concurrency. HPX can not only be used in desktops but also in a wide range of hand-held devices also. With a thriving developer community and liberal open source license, HPX is the first fully functional application of a ParalleX model.

To test the working of HPX, we have made use of a library called **OpenNN**. OpenNN is developed by Artelnics which stands for Open Neural Networks. It is a software library written in C++ programming language. It implements a major area of machine learning, neural networks. Written in C++ for advanced analytics, OpenNN’s main advantage is its high performance. The OpenNN library stands out in terms of execution speed and memory allocation. It is constantly optimized and parallelized in order to maximize its efficiency. It deals with artificial intelligence solutions like regression, classification, forecasting, association and uses sophisticated algorithms and utilities to deal with them. With more than 5000 researchers OpenNN library finds major application in business intelligence, health care and engineering purposes. Some application examples include customer segmentation, churn prevention, early diagnosis of a disease, microarray analysis and other topics like performance optimization and predictive maintenance.

2. HPX IMPLEMENTATION IN MACHINE LEARNING LIBRARIES

2.1. PyTorch Programs

Pytorch, which is an open source machine learning library, primarily developed by **Facebook’s AI Research Lab**, was used during the initial stage of the project. The idea was to use the C++ API of a Machine Learning Library like TensorFlow and use it to test the performance of HPX. But, the TensorFlow C++ API is limited and PyTorch provides a more refined C++ interface. It also has a wide range of applications, as a result of which, it was considered for the testing of HPX.

PyTorch has been written in C++, Python and CUDA. The C++ integration was our main focus since HPX is a C++ library. Our tests mainly focused on implementing HPX into basic machine learning algorithms . We started the testing process with standard PyTorch programs like autograd, mnist and linear regression. The conversions focused on utilising the HPX runtime in these programs. The HPX runtime was successfully applied to these programs, which enabled us to assign work to the cores and run different functions parallely. To run these functions parallely, we used a combination of asynchronous function calls(async) and futures as provided by C++, which is also supported by HPX. Moreover, standard C++ for-loops were also converted into HPX for loops. These parallel algorithms helped us achieve positive performance improvements. The implementation of HPX was done on the following programs:

(1) *Tensor Functions*: This included simple tensor functions and their implementation using HPX.
(2) *Optimization*: The program mimics the working of an optimizer(Adam Optimizer) in Neural Networks.
(3) *Autograd*: This program runs basic autograd operations with some higher order gradient examples and some of the custom autograd functions.
(4) *MNIST*: The Program works on the dataset of hand-written digits and displays the loss and accuracy at the end of each epoch.
(5) *CNN Program -resnet 50*: This program implements ResNET50 (CNN) classifier, which identifies objects in images.
(6) *Linear Regression*: A simple Linear Regression program.
(7) *Logistics Regression*: A simple Logistic Regression program.
(8) *2-Layer Neural Network*: A fully-connected ReLU network with one hidden layer, trained to predict y from x by minimizing squared Euclidean distance.
(9) *Feed Forward Neural Network*: It is a biologically inspired classification algorithm. It takes the input, feeds it through several layers one after the other, and then finally gives the output.

These are a few common programs used by the PyTorch Community and HPX has given a positive performance improvement in most of the cases. More than half of the examples tested during this phase had a performance improvement of over 50%. During this phase of testing, we had developed a converter that converts PyTorch C++ code into HPX implemented PyTorch C++ code. This helped us in converting any given program.

2.2. Converter Tool

The converter is a tool that was developed to convert standard C++ code to HPX implemented C++ code. The contents of a given file are stored in a temporary string and the necessary changes are made and stored into an 'update' string. After all changes have been made, the 'update' string is written back into the file. The methodology of the converter is as follows:

2.2.1. HPX Header Files. The HPX header files are stored in a string and when given a C++ file, the string is written to the beginning of the 'update' string and and the 'update' string is written back into the file.

2.2.2. C++ function calls. A function prototype is a declaration of a function that specifies the function's name and type signature return type), but omits the function body. Thus, we traverse through the entire file character by character to find appropriate user-defined functions and store the function names and their corresponding return types in two separate vectors. Once this has been done, we search for function calls of the functions stored in the vector. If an appropriate function call has been found, we convert the function call to a HPX asynchronous function call and we keep track of the number of parenthesis in order to extract the actual parameters. Once the necessary changes have been done, we store the resulting string in the 'update' string.

Normal Execution	HPX Execution
`int main()`	`int main()`
`{`	`{`
` int num = 10;`	` int num = 10;`
` //function call 1`	` //function call 1`
` func_a();`	` hpx::future<void> f1 =`
` //funtion call 2`	`    hpx::async(func_a);`
` int result = func_b(100);`	` //function call 2`
` //function call 3`	` hpx::future<int> f2 =`
` double res = func_c(num);`	`    hpx::async(func_b,100);`
` return 0;`	` int result = f2.get();`
`}`	` //function call 3`
	` hpx::future<double> f3 =`
	`    hpx::async(func_c,num);`
	` double res = f3.get();`
	` return 0;`
	`}`

Table 1: Difference between normal C++ and HPX function calls

2.2.3. for loops. To achieve this, we traverse through the file and check for the presence of the 'for' keyword. Once found, we check if it is followed by a ' ' or '('. If this condition is satisfied, we have now confirmed the presence of a for-loop. We traverse through the particular line till we encounter a ')' and store the for-loop constraints/conditions in separate variables. We then convert the for-loop into the HPX for-loop and add the updated code to the 'update' string.

Normal Execution	HPX Execution
`int a =10;`	`int a =10;`
`for(int i =1;i<10;i++)`	`hpx::parallel::v2::for_loop_n(`
`{`	`    hpx::parallel::execution::par,1,10,`
`   a+=10;`	`    [&](int i)`
`}`	`   {`
	`      a+=10;`
	`   });`

Table 2: Difference between C++ and HPX for loops

2.2.4. C++ input/output streams. We traverse through the file one line at a time, where we check for the presence of 'std::cout'. If found, we add 'hpx::cout' to the 'update' string and execute the same process till we reach the end of the file. Else, we just add the line to the update string. Finally, we initialise the original file to the 'update' string.

2.3. OpenNN: The Neural Network Library

OpenNN is written in ANSI C++ and can be built on any system. It uses CMake as the build tool. The software model of OpenNN is based on the Unified Modelling Language, which is a visual modelling language used to visualize, specify, construct and document the artifacts of a software system.

2.4. OpenNN File Structure

Opennn follows top-down development. This approach to a given problem begins at the highest conceptual level and works down to the details. The OpenNN library is built on different files put together and every module file is classified under the topic of Classes, Associations, Compositions, Derived classes, Members and methods. These files are installed to the system according to their hierarchy and are then used to train their model. OpenNN uses 5 files for classes, namely,

(1) *DataSet*
(2) *NeuralNetwork*
(3) *TrainingStrategy*
(4) *ModelSelection*
(5) *TestingAnalysis*

The *Dataset* class contains utilities that deal with the processing and treatment of data. The *NeuralNetwork* class combines all the different neural network algorithms.The *TrainingStrategy* class represents the concept of training neural networks in OpenNN . The *ModelSelection* class is used for finding a network architecture with maximum generalization capabilities and the *TestingAnalysis* class contains tools for testing neural networks in learning tasks.

The main classes are associated between two concepts which points to main concepts. These are *TrainingStrategy* which are associated with neural networks and dataset concepts to perform neural network training. *TestingAnalysis* associates Dataset and *NeuralNetwork* to evaluate a neural network over a dataset.

In OpenNN the concepts of *Dataset, NeuralNetwork, ModelSelection* and *TestingAnalysis* are high level structures and are composed of different elements. Dataset is not composed of any other classes. It's a high-level class where all the methods are controlled by it. *NeuralNetwork* is composed of many other classes, some of which are *PerceptronLayer, ScalingLayer, ConvolutionaLayer* and *PoolingLayer*. The *TrainingStrategy* class contains the *LossIndex* abstract class which represents the concept of error term and the *OptimizationAlgorithm* abstract class represents the training algorithm for a neural network.

The classes *MeanSquareError, CrossEntropyError, MinkowskiError, NormalizedSquaredError, SumSquaredError, WeightedSquareError* are derived from *LossIndex*, all of which inherit the characteristics of *LossIndex*. However, each of these classes also introduce new features such as the definition of their own error. Likewise *GradientDescent* and *AdaptativeMomentEstimation* are some of the classes which have been inherited from the abstract class *OptimizationAlgorithm*.

For some instances of classes there exists a relationship called a member or an attribute, and a method or operation is associated with these classes. Moreover, this library uses its examples to train itself .

2.5. HPX BackEnd in OpenNN

Firstly, changes were made to CMakelists.txt to include HPX library. The main aim was to execute programs on the HPX runtime .The HPX runtime was successfully applied to these programs, which enabled us to assign work to the cores and run different functions parallely. The use of HPX parallel algorithms helped us achieve positive performance improvements. Changes were made to CMakeLists in the opennn directory to create a shared CXX hpx_library.

The OpenNN github repository consisted of many examples to which HPX was added. The implementation of HPX was done on the following example programs: *breast_cancer, airfoil_self_noise, logical_operations, airline_passengers,iris_plant and simple_function_regression*.

Programs like breast_cancer , airline_passenger and many others used the function *perform_training()*, which is defined in the module *adaptive_moment_estimation.cpp*. This module represents the backend to implement a neural network and has been

utilised by multiple examples. Hence, HPX has been implemented in this particular module by the use of HPX parallel algorithms and HPX output streams. This class file is considered as a derived class file. It is derived from the abstract class TrainingAlgortihm. This file along with other similar files are responsible for the training and optimization of the OpenNN library and helps in providing better results for this library. The adaptive_moment_estimation.cpp is important and the functions included in the file play a very important role in optimization of the whole library.
The function call to *perform_training()* present in the main.cpp file of airline_passengers example.

```
const OptimizationAlgorithm::Results training_strategy_results =
    training_strategy.perform_training();
```

The parallelizing of for loop in the function perform_training() in the file *adaptive_moment_estimation.cpp*

```
hpx::parallel::v2::for_loop_n(
       hpx::parallel::execution::par,0,batches_number,[&](size_t iteration){
       ...
});
```

Similarly, it has been observed that functions like *get_input_variables_names()* , *get_target_variables_names()* were frequently called in the example programs. These functions were defined in *dataset.cpp* and changes were made to the file to implement HPX. In Object Oriented Programming, the concepts used are presented through the class files. dataset.cpp is considered as a class file. This class processes and works with the treatment of data. Along with neuralnetwork.cpp, the dataset.cpp file plays a very important role and provides basic functionality to the OpenNN library. Considered as a higher-level-class file, dataset.cpp is dependent on many prominent lower-level-class files. The trainingstrategy.cpp file works with the neuralnetwork.cpp and dataset.cpp files to help them perform neural network training. The modelselection.cpp file acquires the results from trainingstrategy.cpp and helps determine the best model to get better accuracy and results. Finally, the trainingstrategy.cpp file works on the evaluation of neuralnetwork.cpp over a data set.
The function call to *get_input_variables_names()* and *get_target_variables_names()* present in the main.cpp file of airfoil_self_noise example.

```
const Vector<string> inputs_names = data_set.get_input_variables_names();
const Vector<string> targets_names = data_set.get_target_variables_names();
```

The parallelizing of for loop in the function get_input_variables_names() in the file *dataset.cpp*

```
hpx::parallel::v2::for_loop_n(
       hpx::parallel::execution::par,0,input_columns_indices.size(),[&](size_t i){
      size_t input_index = input_columns_indices[i];

      const Vector<string> current_used_variables_names =
          columns[input_index].get_used_variables_names();

      input_variables_names.embed(index, current_used_variables_names);

      index += current_used_variables_names.size();
});
```

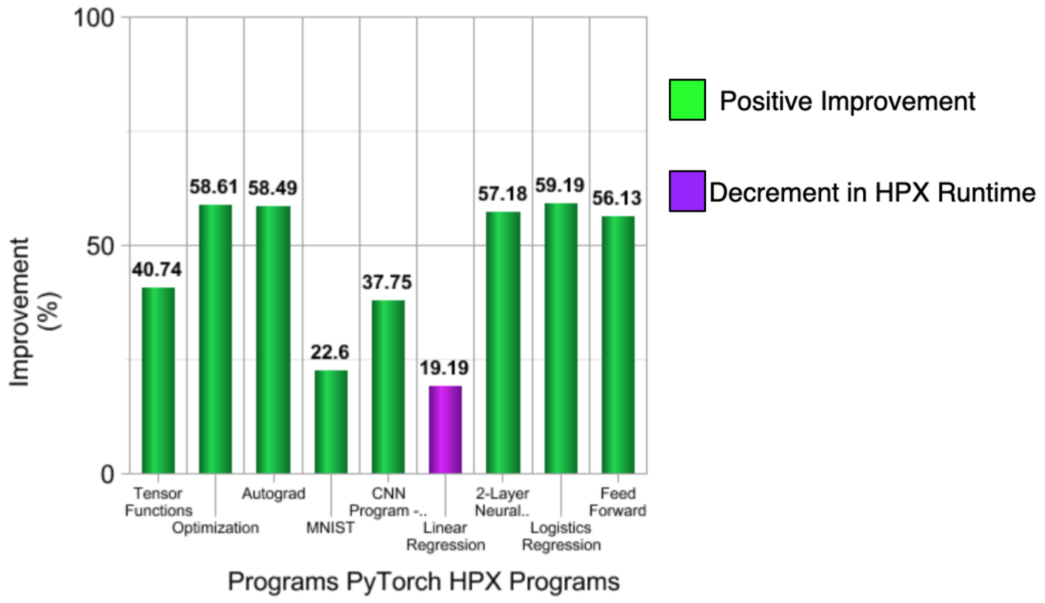
The parallelizing of for loop in the function get_target_variables_names() in the file *dataset.cpp*

```
hpx::parallel::v2::for_loop_n(
        hpx::parallel::execution::par,0,target_columns_indices.size(),[&](size_t i){
        size_t target_index = target_columns_indices[i];

        const Vector<string> current_used_variables_names =
            columns[target_index].get_used_variables_names();

        target_variables_names.embed(index, current_used_variables_names);

        index += current_used_variables_names.size();
});
```

More changes were made to different functions and thus a reduction in execution time was noticed.

3. RESULTS

3.1. PyTorch Results

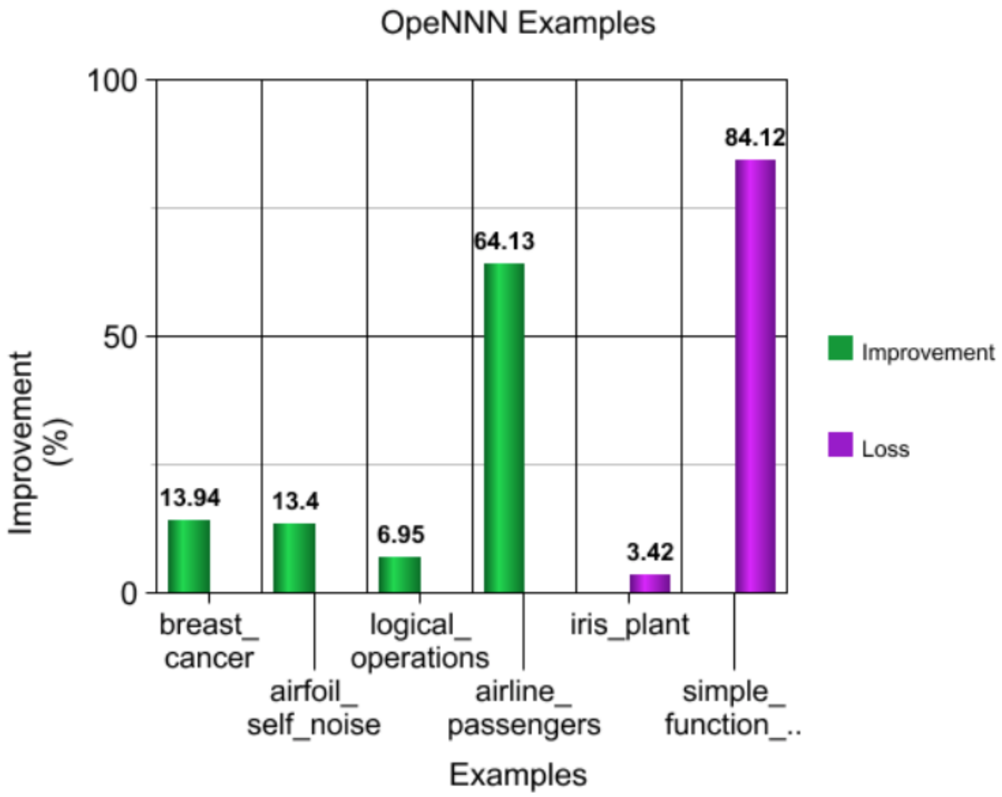


From the above graph, it can be seen that eight out of the nine programs show a positive performance improvement. More than half of the examples tested gave more than 50% improvement and they are very prominent examples. The use of HPX asynchronous function calls and HPX for-loops combined with the application of the HPX runtime system enabled individual tasks to run directly on the system cores. The performance can be further improved by the use of HPX lightweight threads. However, in case of linear regression, a loss in the performance can be observed. This is because the get() function has been called before another asynchronous function call. Hence, the system has to wait till the required result has been fetched and once it's done, the function is called asynchronously. This results in a time delay which explains the increased execution time and consequent loss in performance. This may be considered as

an edge case of the HPX runtime system.
Code present in Linear Regression which is responsible for the delay.

```
hpx::future<std::string> f2 = hpx::async(poly_desc,fc->weight.view({-1}),
    fc->bias);
hpx::cout << "==> Learned function:\t"<< f2.get() << hpx::endl;
hpx::future<std::string> f3 = hpx::async(poly_desc, W_target.view({-1}), b_target);
hpx::cout << "==> Actual function:\t"<< f3.get() << hpx::endl;
```

3.2. OpenNN Results



From the above graph, it can be seen that most of the examples show a positive performance improvement. The loss in performance is observed because the program as such is very small(100 lines) and hardly utilises any HPX functionality. This is because the functions were called within other module files as a result of which they could not be called asynchronously. Moreover, the execution time for simple_function_regression is around 1 second and hence a great improvement in the execution time could not be achieved.

4. LIMITATIONS

(1) HPX does not support the use of C++ templates depending on the data type of the template.
(2) HPX asynchronous function calls cannot be made for functions that are defined for structures.

(3) The HPX for-loop does not support the concept of jump statements.
(4) HPX requires the installation of its dependencies(eg: HWLOC, BOOST) without which HPX programs cannot be executed.
(5) The use of HPX asynchronous function calls may lead to a race condition for different programs on different systems.(eg : MNIST program)
(6) One should be familiar with cmake and CMakeLists to execute HPX programs.
(7) OpenNN is a vast software library and hence, only a few module files have been converted to HPX.
(8) Depending on the system, one might face "Template" errors due to the system configuration.
(9) The program to be converted should be well written with functions otherwise HPX might not be able to perform well.

5. CONCLUSIONS

HPX is a modern open-source C++ library which can be used to parallelize the backend of other libraries as well. And so far, the results we have seen are very promising. It has made the PyTorch programs very efficient and also has provided a way to parallelize the backend of OpenNN library. The results we present demonstrate a high degree of performance improvement based on a very portable implementation which is in accordance to the latest C++ Standards. This library stands out because of its better execution speed and because of its high resource utilization. Some of the cases where HPX failed to perform can be considered as edge cases. We believe that this project shows that HPX can be used to provide positive results in already existing Machine Learning libraries.

6. FUTURE DIRECTIONS

— Improving the functionality and performance of the programs by creating own HPX threads and using HPX resource partitioner to create thread pools and assigning them with tasks.
— Making the convertor more robust and detection of edge cases.
— Updating the converter tool to make changes to the CMakeLists.txt.
— Implementing HPX backend for PyTorch , mlpack and other libraries.
— Extending the convertor to implement other parallel algorithms of HPX.
— Some of the functions in OpenNN have been parallelized and hence other functions have to be parallelized using hpx to give better results.

7. INDIVIDUAL CONTRIBUTIONS

7.1. AMRITESH

— Working on conversion of PyTorch examples.
  — Autograd
  — MNIST
  — Linear Regression
— Working on the converter
— To identify the function declaration in a PyTorch Program and make their function calls into asynchronous HPX async calls.
— Working on the BackEnd of the OpenNN files and conversion CMakeLists.txt and addition of HPX in some backend files like adaptive_moment_estimation.cpp as well as examples like airline_passengers and airfoil_self_noise.

7.2. ASHWATH KRISHNAN

— Working on conversion of PyTorch examples.

  - Tensor Functions
  - Logistic Regression
  - 2-Layer Neural Network
- Working on the converter
- To identify the a for loop and convert it into HPX for_loop.
- Working on the BackEnd of the OpenNN files and addition of HPX in some backend files and also making for loop changes in files like adaptive_moment_estimation.cpp and examples like simple_function_regression and breast_cancer.

7.3. Srikar S

- Working on conversion of PyTorch examples.
  - Optimization
  - CNN Program -resnet 50
  - Feed Forward Neural Network
- Researching on the backend of PyTorch and OpenNN
- Working on implementing HPX in PyTorch backend
- Working on the BackEnd of the OpenNN files and addition of HPX in some backend files and also making changes in files like dataset.cpp and examples like logical_operations and iris_plant.

8. REFERENCES

(1) HPX – A Task Based Programming Model in a Global Address Space (http://stellar.cct.lsu.edu/pubs/pgas14.pdf)
(2) STEllAR-GROUP HPX Github (https://github.com/STEllAR-GROUP/hpx)
(3) Sessions hosted by CSCS on HPX (https://github.com/STEllAR-GROUP/tutorials/tree/master/cscs2016)
(4) 20160929 0900 CSCS HPX Introduction to HPX Part 1 Biddiscombe video (https://www.youtube.com/watch?v=NToOo-T3Q3w&t=630s)
(5) 20160929 1045 CSCS HPX Introduction to HPX Part 2 Biddiscombe, Heller video (https://www.youtube.com/watch?v=ZZxvqJszLxU&t=147s)
(6) Stellar Group website (https://stellar-group.org/libraries/hpx/)
(7) Basic Notes on HPX (http://stellar.cct.lsu.edu/projects/hpx/)
(8) HPX Notes/Slides (https://stellar-group.github.io/tutorials/hlrs2019/session2/#1)
(9) Installation of HPX on macOS (http://stellar.cct.lsu.edu/files/hpx_0.9.7/html/hpx/tutorial/getting_started/macos_installation.html)
(10) HPX examples (https://github.com/STEllAR-GROUP/tutorials/tree/master/examples)
(11) HPX website with information regarding ”Writing single-node HPX applications” (https://stellar-group.github.io/hpx/docs/sphinx/branches/master/html/manual/writing_single_node_hpx_applications.html)
(12) Stellar Group HPX libraries (https://stellar-group.github.io/hpx/docs/sphinx/latest/html/libs/affinity/docs/index.html)
(13) PyTorch C++ installation (https://pytorch.org/cppdocs/)
(14) PyTorch Github (https://github.com/pytorch/pytorch)
(15) PyTorch Examples (https://github.com/pytorch/examples/tree/master/cpp)
(16) PyTorch Examples/Tutorials (https://pytorch.org/tutorials/beginner/pytorch_with_examples.html)
(17) Basic libTorch Examples (https://github.com/Maverobot/libtorch_examples/tree/master/src)
(18) PyTorch BigBallon Resnet example (https://github.com/BIGBALLON/PyTorch-CPP)
(19) PyTorch Documentation (https://pytorch.org/docs/stable/index.html)

(20) OpenNN Github (https://github.com/Artelnics/opennn)
(21) OpenNN website (https://www.opennn.net/)
(22) OpenNN Documentation (https://www.opennn.nethttps//www.opennn.net/documentation/)

9. APPENDIX

data_set.cpp is a module file whose functions are utilised in almost all the OpenNN examples. However, the module as such is very large(9000 lines). Hence, the following code snippet depicts the implementation of HPX in a few functions in data_set.cpp.

```
Vector<size_t> DataSet::get_instances_uses_numbers() const
{
    Vector<size_t> count(4, 0);
    const size_t instances_number = get_instances_number();
    hpx::parallel::v2::for_loop_n(
        hpx::parallel::execution::par,0,instances_number,[&](size_t i)
    {
        if(instances_uses[i] == Training)
        {
           count[0]++;
        }
        else if(instances_uses[i] == Selection)
        {
           count[1]++;
        }
        else if(instances_uses[i] == Testing)
        {
           count[2]++;
        }
        else
        {
           count[3]++;
        }
    });
    return count;
}

Vector<size_t> DataSet::get_training_instances_indices() const
{
    const size_t instances_number = get_instances_number();
    const size_t training_instances_number = get_training_instances_number();
    Vector<size_t> training_indices(training_instances_number);
    size_t count = 0;
    hpx::parallel::v2::for_loop_n(
        hpx::parallel::execution::par,0,instances_number,[&](size_t i)
    {
       if(instances_uses[i] == Training)
       {
          training_indices[count] = static_cast<size_t>(i);
          count++;
       }
    });
    return training_indices;
}

Vector<size_t> DataSet::get_selection_instances_indices() const
```

```
{
    const size_t instances_number = get_instances_number();
    const size_t selection_instances_number = get_selection_instances_number();
    Vector<size_t> selection_indices(selection_instances_number);
    size_t count = 0;
    hpx::parallel::v2::for_loop_n(
        hpx::parallel::execution::par,0,instances_number,[&](size_t i)
    {
       if(instances_uses[i] == Selection)
       {
          selection_indices[count] = i;
          count++;
       }
    });
    return selection_indices;
}

size_t DataSet::get_training_instances_number() const
{
    const size_t instances_number = get_instances_number();
    size_t training_instances_number = 0;
    hpx::parallel::v2::for_loop_n(
        hpx::parallel::execution::par,0,instances_,[&](size_t i)
    {
       if(instances_uses[i] == Training)
       {
          training_instances_number++;
       }
    });
    return training_instances_number;
}

void DataSet::set_k_fold_cross_validation_instances_uses(const size_t& k, const
     size_t& fold_index)
{
    const size_t instances_number = get_instances_number();
    const size_t fold_size = instances_number/k;
    const size_t start = fold_index*fold_size;
    const size_t end = start + fold_size;
    split_instances_random(1, 0, 0);
    hpx::parallel::v2::for_loop_n(
        hpx::parallel::execution::par,start,end,[&](size_t i)
    {
        instances_uses[i] = Testing;
    });
}

void DataSet::set_default_columns_names()
{
   const size_t size = columns.size();
   if(size == 0)
   {
        return;
   }
   else if(size == 1)
   {
        return;
   }
   else
   {
       size_t input_index = 1;
       size_t target_index = 2;
       hpx::parallel::v2::for_loop_n(
        hpx::parallel::execution::par,0,size,[&](size_t i)
       {
           if(columns[i].column_use == Input)
           {
               columns[i].name = "input_" + std::to_string(input_index);
               input_index++;
           }
           else if(columns[i].column_use == Target)
           {
               columns[i].name = "target_" + std::to_string(target_index);
               target_index++;
           }
       });
   }
}
```