Adoption of High Performance Computing in Neural Designer

Judit Gimenez*, Fernando Gomezb, Roberto Lopezb

* Barcelona Supercomputing Center, Spain
b Artelnics S.L., Spain

Abstract
Artelnics develops the professional predictive analytics solution called Neural Designer. It makes intelligent use of data by discovering complex relationships, recognizing unknown patterns, predicting actual trends or finding associations. The new OpenMP and MPI version of the code allows Artelnics to build predictive models in computer instances with many virtual cores and in supercomputing clusters, respectively. The current version of the code reports efficiencies close to 90% for both the MPI and the OpenMP parallelizations. Neural Designer now is capable of analysing bigger data sets in less time, providing Artelnics customers with results in a way previously unachievable.

Introduction
The predictive analytics market is undergoing an impressive growth. Indeed, organizations that incorporate that technique into their daily operations not only better manage the present, but also increase the probability of future success.

Artelnics develops the professional predictive analytics solution called Neural Designer [1]. It makes intelligent use of data by discovering complex relationships, recognizing unknown patterns, predicting actual trends or finding associations. Neural Designer out-stands in terms of usability, functionality and performance.

Current technology lacks advanced model selection techniques, and usually requires many computational resources. The main challenge for Neural Designer is to include a framework capable of untangling complex interactions in big data sets. In order to do that, the software must achieve high performance by means of parallel processing.

The users of the solution are professional data scientists, who work at analytics departments of innovative companies, consulting firms specialized in analytics or research centres. Neural Designer will be capable of analysing bigger data sets in less time, providing Artelnics customers with results in a way previously unachievable.

Within the framework of the Performance Optimisation and Productivity Centre of Excellence (https://pop-coe.eu) BSC analysed the OpenNN code [2] originally parallelised with OpenMP. As reported by the POP service, the parallel versions of OpenNN and Neural Designer are running more than 5 times faster than the serial versions in a desktop computer [3].

The next step developed within the framework of POP and SHAPE has been the implementation of shared and distributed memory parallelisation by means of OpenMP and MPI. The new version of the code allows Artelnics to build predictive models in computer instances with many virtual cores and in supercomputing clusters, respectively.

* Corresponding author. E-Mail address: judit@bsc.es
Background

OpenNN is able to make a lot of analysis tasks with the data, but the most important is the training of the neural network. This is an iterative algorithm. Several times in each iteration the library must calculate the outputs of the neural network and the gradient of the loss function.

At the same time, those calculations are made iteratively through the instances of the data of the user. Usually a simple data set has several hundreds of instances, so this procedure can produce a lack of performance in the execution.

The Quasi Newton method is used for the model training in this study. It is one of the most used algorithms in this library. The algorithm follows the Newton method, but without calculating the exact inverse of the Hessian matrix. Instead of the exact inverse Hessian, the Quasi Newton method uses an approximation of it using different formulas. These formulas are computed with several operations with matrices. The next figure shows the flow diagram of this algorithm.

![Flow diagram of the Quasi Newton algorithm.](image)

Before the MPI implementation the OpenNN library was only parallelized with OpenMP. This implementation was only made in the most used functions in a regular execution. Merely the calculation of the gradient and the calculation of the outputs was parallelized. The operations with matrices were only parallelized with CUDA, but those did not use the multi thread parallelization in the case when a NVIDIA GPU was not available in the system.

A previous analysis of performance was made in the POP program. Despite of this issue with the OpenMP parallelization, the report showed more than 90% of parallel efficiency for the multithread parallelization. Though the maximum number of threads used was just eight.

Nowadays, the stored data can surpass the previous limits with the world of the Big Data. All these data can also be used for analysis by some users. The OpenNN library only utilized the memory of the user machine. With the MPI implementation, we will be able to distribute the data and allow the software to use bigger amounts of data.

Technical approach

The first MPI+OpenMP prototype was analysed with the BSC performance tools [4] in MareNostrumIII. Before the SHAPE project was being evaluated, the prototype was decommissioned within the scope of POP. We also used that opportunity to evaluate the scaling of the OpenMP up to 16 threads.

The OpenMP version had a very good parallel efficiency for the different executions with a small variability that was strongly related with the global load balance. The analysis detected a degradation in the computations when scaling, because when we increase the number of threads we are increasing the load of the node. That was reflected as a reduction of the clock frequency and the IPC.
The preliminary hybrid MPI+OpenMP version had a very poor scaling. The traces allowed to identify that the main problem was related with a region, which was being replicated and represented 35% of the main computing loop, when running with 1024 cores. Figure 2 plots with Paraver the instructions counter for two different runs increasing the number of MPI ranks. The blue regions correspond to the replicated code that does not reduce the number of instructions when scaling. The analysis also detected degradations in the initialization phase when increasing the number of cores related with serialized chains that should be eliminated to target very large scales. These inputs where used by Artelnics to improve the original prototype.

Within the SHAPE project we have evaluated the MPI+OpenMP version running in the Marconi Broadwell partition. This process has been iterative between Artelnics and BSC, interchanging new improved versions of the code and results for the performance analysis. The main sources for degradation where related to load unbalance. In some cases, the unbalance was caused by the code itself due to a non-balanced amount of work between MPI processes or between OpenMP threads. As an example Figure 3 displays the duration of the large computing regions in an execution of 16 MPI ranks with 8 OpenMP threads each. We can see that along the whole execution the same processes have constantly a darker or lighter colour. The uniform distance between the darker areas seems to indicate that the structured unbalance is either correlated with the code or with the process mapping. The instruction counter revealed this behaviour was due to a structured unbalance of the work.

Another source of unbalance is caused by the resources policies of CINECA that by default share nodes between different jobs and users. In Figure 4 we correlate the average IPC achieved by each process versus the mapping on the different nodes. In this picture the columns correspond to the 4 nodes allocated to the job and the rows to the different processes/threads launched. We can see that the higher ranks that are mapped with fewer processes have a darker colour which means they improve the IPC as compared with the nodes where more processes are mapped. Also, we can see that the first process of each node is coloured in orange, i.e. they get an even better IPC. We do not know the reason for that, but apparently the first core of each node use to get better IPC in all the executions that we have analysed. Despite we were able to detect this unbalance, the impact in the execution is usually low, sometimes negligible, and usually lower than 5%.

Figure 2: Comparing the executions 16x4 (top) and 256x4 (bottom) eliminating the initialization phase

Figure 3: Structured unbalance detected by the performance analysis

Figure 4: IPC correlation with mapping
This MPI implementation has been made targeting to make the library more generic and user friendly. The high-level functions remain with the same interface and the MPI functions are restricted to the lowest level instructions. The users have to add few “if statements” in their previous codes.

The most parallelized functions are those that are the most frequently called during a normal execution. In this part the code was already parallelized with OpenMP. Now, the implementation performs the most computationally difficult tasks in different nodes each with different data.

The main objective to introduce the MPI parallelization in the OpenNN library was to enable loading bigger data and perform the required analysis of them. Thus, that big data set must be separated in different subsets for the nodes. This causes the complexity of the initialization phase, related to the necessity to exchange the data information with all the MPI nodes.

The library has to be usable with all possible data sizes. Some synchronization problems may occur when the number of nodes is greater than the number of instances of the data set, since some nodes will not perform any operation. Consequently, an implementation to have several synchronizations with all the nodes makes the code usable for different amounts of analysed data.

Results

This section presents the results for the latest version that was analysed using the Broadwell partition in Marconi. The jobs were executed using the exclusive mode in order to eliminate any perturbation either due to a potential sharing of the resources or an unbalanced mapping. After confirming a uniform behaviour along time, the analysis was focused on a representative region of the iterations, eliminating the initialization phase.

The final evaluation targeted two objectives:

- to evaluate the combination of MPI+OpenMP and
- to study the scaling of the MPI implementation.

For the first goal we used one node of Marconi with 32 cores and different configurations that are included in the next table together with the execution time and the efficiency on the selected region.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>MPI ranks</th>
<th>OpenMP threads</th>
<th>Time (seconds)</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>1</td>
<td>18.10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>2</td>
<td>18.84</td>
<td>0.96</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>4</td>
<td>19.38</td>
<td>0.93</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>20.09</td>
<td>0.90</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>16</td>
<td>21.06</td>
<td>0.86</td>
</tr>
</tbody>
</table>
Despite the small time degradation when increasing the number of threads, the analysis shows a very good parallel efficiency for all the configurations (between 87% and 98%). The small degradation with more threads can be attributed to a small reduction in the IPC when the number of threads is increased and to a small imbalance between the threads.

Figure 5 shows the reported IPC reduction on the different computing regions based on a clustering analysis [5]. The scatter plots display the regions based on their performance (IPC on the X axis) and computation (number of instructions on the Y axis). Except for the yellow cluster, we can see how most of the clouds move to the left (less IPC) in the configuration with more threads. This reduction of the IPC might be related to the shared memory access, though it was not deeply analysed due to its small impact.

Figure 5: Clustering analysis for 32x1 (left) and 2x16 (right)

Figure 6 captures a zoom in the timelines displaying the parallel functions for the fastest and slowest configurations, within the same time interval. We can see that the duration of the blue parallel function is very similar in both executions (a little bit larger with 2x16) while the red one is also enlarged in the two MPI ranks generating a black gap (idle time) in the slave threads. The recommendation would be to limit to no more than 4 or 8 threads per MPI rank depending on the node size.

Figure 6: Parallel functions view for the configurations 32x1 (top) and 2x16 (bottom)

The analysis of the MPI scaling was targeted setting OMP_NUM_THREADS to 1, as a way to stress the MPI part with lower number of processes. The maximum number of nodes was limited to 4 to be able to use the debug queue. The next table collects the configurations analysed, the execution time and the speed-up achieved on the selected region.
The reported speed-ups are quite good since the ideal speed-up with 128 ranks with respect to the execution of 32 ranks would be 4 and the application achieves an 87%.

Despite these results, the analysis of the MPI part identified space for potential improvements. There are frequent calls to MPI Comm rank, MPI Comm size, MPI Comm create and MPI Comm free. More than 3500 communicators are created in 4 minutes of execution. Such high ratio seems to indicate it would be possible to reuse them. There are two calls to MPI Comm rank and one to MPI Comm size per MPI Comm create call that would be eliminated if the communicators were reused. These calls represent 0.5% of the execution and goes up to 2% with 128. So, despite there are not major problems on the analysed scale, these issues may become more important for executions with thousands of MPI ranks.

A second potential improvement was identified analysing the sequences of MPI calls. In many of the communication phases the following sequence is repeated with very small computations between these calls.

```
MPI_Reduce()
MPI_Barrier()
MPI_Barrier()
MPI_Comm_free()
MPI_Barrier()
MPI_Bcast()
```

To allow executions at large scale, it would be important to eliminate the barriers. In fact, the sequence seems to indicate that the barriers are not needed. The MPI processes can progress independently from the MPI Reduce to the MPI Bcast, unless there is an I/O that requires the synchronization to serialize the access.

The collected traces also reflect some variability or noise in the MPI calls. As example, the MPI Comm create calls that typically last 230 microseconds have also executions in the order of 700 microseconds and few of them in the order of 1200 microseconds. Even the MPI Comm rank call - that should be a local call - sometimes reports durations more than 10 times of its usual duration. Reducing the suggested calls would also make the application less sensible to the MPI noise.

**Future work**

We have reached a good approach for the first phase of a MPI implementation. Nevertheless, this code can be improved with several changes. Besides, Artelnics’ users are data scientist that usually have no easy access to a supercomputer, so the objective is to make Neural Designer capable to run in cloud platforms with a variety para of parallelizations.

First, the high number of synchronizations and barriers allow the library to be usable for different sizes of data with a low number of changes in the code. These extra actions are the most relevant in the lack of speed-up. So, the most important result of this paper must be an optimization of the code, examining these problems and trying to solve the slowdown of the performance.

The testing of this implementation has been carried out with an artificial example of a regular execution with OpenNN. This study shows a way to follow for different cases, so this implementation must be tested also for a real data analysis. The final objective of Artelnics is to apply the proposed improvements in the Neural Designer software, with the aim to enhance its applicability for the end users.

Finally, with the advent of cloud computing in recent years, even the average user can have an access to a great machine, being able to make a multi-processor system. Neural Designer is on the AWS marketplace and Artelnics is going to have its software in Microsoft Azure platform, too, thus enabling the user to perform the analyses on more efficient machines than their own ones. The MPI parallelization enables and facilitates analysing bigger data and substantially speeds up the execution.
Conclusions

The main objective to introduce the MPI parallelization in the OpenNN library was to enable to load bigger data and perform the required analysis with the data. This objective has been successfully accomplished within this SHAPE project thanks to a close and iterative cooperation between Artelnics and BSC. The continuous feedback about the improvements implemented by the code owner, guided the next optimizations and resulted in a very efficient parallel MPI+OpenMP version with efficiencies around 90% in the analysed scale.

The final evaluation has been done in the Marconi Broadwell partition and targeted both, to evaluate the combination of MPI+OpenMP and to study the scaling of the MPI implementation.

Besides, the MPI distribution of work makes OpenNN capable of computing bigger data set, while all the instances can be mapped in different nodes. Hence, the main node does not need to load all the data filling all the memory.

The new version of the code allows Artelnics to build predictive models in computer instances with many virtual cores as well as in supercomputing clusters. Neural Designer is now capable of analysing bigger data sets in less time, providing Artelnics customers with results in a way that was previously unachievable.

References


Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 730913.