Task-based Acceleration of Bidirectional Recurrent Neural Networks on Multi-core Architectures

Robin Kumar Sharma
Computer Science Department
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
robinkeralaam@gmail.com

Marc Casas
Computer Science Department
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
marc.casas@bsc.es

***Abstract*—This paper proposes a novel parallel execution model for Bidirectional Recurrent Neural Networks (BRNNs), B-Par (Bidirectional-Parallelization), which exploits data and control dependencies for forward and reverse input computations. B-Par divides BRNN workloads across different parallel tasks by defining input and output dependencies for each RNN cell in both forward and reverse orders. B-Par does not require per-layer barriers to synchronize the parallel execution of BRNNs. We evaluate B-Par considering the TIDIGITS speech database and the Wikipedia data-set. Our experiments indicate that B-Par outperforms the state-of-the-art deep learning frameworks TensorFlow-Keras and Pytorch by achieving up to $2.34\times$ and $9.16\times$ speed-ups, respectively, on modern multi-core CPU architectures while preserving accuracy. Moreover, we analyze in detail aspects like task granularity, locality, or parallel efficiency to illustrate the benefits of B-Par.**

***Index Terms*—Deep neural network (DNN), Bidirectional recurrent neural networks (BRNNs), Long-short term memory (LSTM), Gated Recurrent Units (GRU), Task Parallelism**

I. Introduction

Bidirectional Recurrent Neural Networks (BRNNs) [1] are an evolution of the well-known Recurrent Neural Networks (RNNs) [2]. BRNNs are composed of two parallel RNN models: the first model processes input data in forward order, while the second RNN does so in reverse order. This combination of forward and reverse input data processing allows BRNNs to simultaneously capture future and past information for each time instance, which improves the accuracy over unidirectional RNNs. BRNNs are a popular choice for Automatic Speech Recognition (ASR) [3], language translation [4], handwriting recognition [5], medical events detection tasks [6] and image captioning [7]. Furthermore, BRNNs have been widely used in combination with convolutional neural networks (CNNs) [8], [9]. With the advent of attention and transformer models, BRNNs are used as complimentary to attention and transformer models [10], [11] targeting scenarios like language understanding [12], name entity recognition [13], and recommendation systems [14]. In certain scenarios, BRNNs outperform transformer models [15], [16]. BRNNs are a fundamental building block of emerging deep learning workloads.

BRNNs high accuracy comes at the cost of a large number of training parameters and very complex data and control dependencies. These dependencies span across the forward and reverse orders of input processing and complicate the parallel execution of BRNN workloads. Indeed, state-of-the-art deep learning frameworks apply per-layer barriers that produce severe CPU starvation and seriously limit the performance of BRNN workloads on a large number of CPU cores [17].

We propose B-Par, a novel parallel execution model for Bidirectional Long Short Term Memory (BLSTMs) [18] and Gated Recurrent Units (BGRUs) [19] networks. B-Par conceives BRNN workloads as a computational graph where nodes represent computation and edges identify data and control dependencies between them [20]. A run-time system software orchestrates the parallel execution of BRNNs across multi-core CPU devices by scheduling computing pieces as soon as their data or control dependencies are fulfilled. B-Par significantly improves the performance of state-of-the-art deep learning frameworks since it does not need per-layer barriers to synchronize the parallel execution. B-Par does not require the programmer to express the parallel execution schedule explicitly at the source code level, since it is managed at run-time by the system software. Programming environments like OpenMP [21] and OmpSs [22] support the definition of input and output dependencies between tasks and the dynamic management of them. We implement B-Par using OmpSs.

B-Par currently focuses on multi-core CPU systems. While GPUs match with the computation requirements of the DNN training workloads, many-core CPUs also deliver high floating-point operations per second (flop/s) rates and constitute fundamental building blocks of high-performance computing clusters [23], [24]. For example, the Fugaku system [25], which is ranked first by the November 2021 Top 500 list [26] and entirely based on many-core CPU chips, delivers 2.78 Tflop/s per socket while the Summit [27] system delivers 5.37 Tflop/s per GPU. The importance of efficiently using multi-core CPUs to process deep learning frameworks is also motivated by the low latency, they display for small batch sizes and the large number of many-core CPUs that would otherwise be idle during off-peak periods of infrastructures like cloud computing servers or high-performance computing clusters. For example, FBLearner [28] of Facebook uses many-core CPUs to run DNN training and inference workloads. Also, leading companies use CPUs to run real-time inference workloads [17], [29], [30] in contexts like mobile devices [31] and some extreme environments, e.g., space and defense

http://dx.doi.org/10.1109/IPDPS53621.2022.00096

industries [32]. B-Par and the ubiquity of multi-core CPUs have the potential to impact many RNN users.

The core contributions of this work are threefold:

- We propose B-Par, a parallel execution model for BRNN workloads. B-Par exploits model parallelism and relies on source-code annotations indicating input and output dependencies across different BRNN compute kernels. B-Par does not need per-layer barriers to synchronize the parallel execution of BRNN workloads.
- We compare B-Par with state-of-the-art implementations of bidirectional LSTMs and GRUs on two high-end computing systems considering the TIDIGITS speech database [33], the Wikipedia data-set [34], and a wide range of model parameters. B-Par reaches performance speed-ups up to $2.34\times$ in comparison to the most recent implementation of the state-of-the-art Keras-TensorFlow-2.3.0 deep learning framework [35], [36], and $9.16\times$ in comparison to PyTorch-1.7.1 [37].
- We thoroughly analyze B-Par locality-awareness, task granularity, and memory consumption for different BRNN models. This analysis demonstrates the benefits of the B-Par execution model.

II. Background on BRNN

Bidirectional RNNs [1] are an evolution of traditional RNNs that process input data in both forward and reverse directions with two separate unidirectional RNNs. While conventional RNNs and their variants can only use the previous context, BRNNs exploit past and future information. BRNNs use the basic RNN unit and its variants LSTM [18] and GRU [19] to carry out their predictions. A BRNN model uses two sets of weights and biases, one for forward order and one for reverse order input processing.

Fig. 1 shows a 3-layer deep BRNN model with a sequence length of three. BRNNs are composed of a unidirectional RNN model for forward order input processing and another unidirectional RNN model for reverse order input processing, as shown in Fig. 1. Forward order uses inputs in sequence from 1 to n, whereas the reverse order uses the input sequence from n to 1. Each square cell is composed of either an LSTM [38] or a GRU cell [19]. Equations (1)-(6) define the computations involved in each LSTM cell, and previous work [39] contains detailed descriptions of all parameters.

$$f_t = sigm(W_f * [X_t, H_{t-1}] + B_f) \quad (1)$$
$$I_t = sigm(W_i * [X_t, H_{t-1}] + B_i) \quad (2)$$
$$\bar{C}_t = tanh(W_c * [X_t, H_{t-1}] + B_c) \quad (3)$$
$$O_t = sigm(W_o * [X_t, H_{t-1}] + B_o) \quad (4)$$
$$C_t = f_t \odot C_{t-1} + I_t \odot \bar{C}_t \quad (5)$$
$$H_t = O_t \odot \tanh(C_t) \quad (6)$$

Equations (7)-(10) define GRUs cell computations:

$$Z_t = sigm(W_z * [X_t, H_{t-1}] + B_z) \quad (7)$$
$$R_t = sigm(W_r * [X_t, H_{t-1}] + B_r) \quad (8)$$
$$\bar{H}_t = tanh(W_h * [X_t, R_t \odot H_{t-1}] + B_h) \quad (9)$$
$$H_t = Z_t \odot \bar{H}_t + (1 - Z_t) \odot H_{t-1} \quad (10)$$

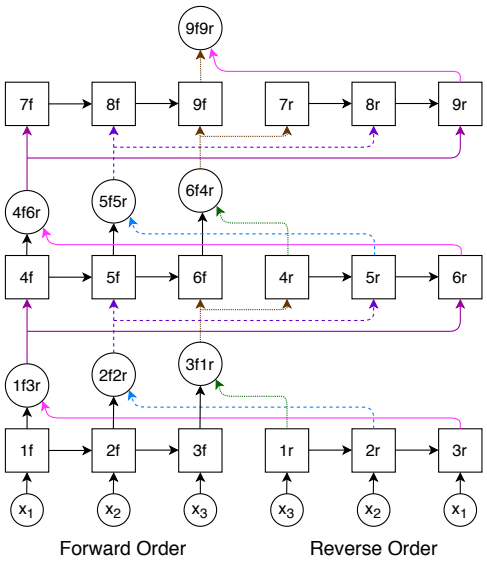

Fig. 1: Many-To-One BRNN model

Each forward and reverse order RNNs have a recurrent hidden state whose activation value at time t, H_t, depends on the hidden state value at time $t-1$, H_{t-1}. These cyclic dependencies complicate the practical implementation of RNNs and motivate the need for unrolling RNNs to form a directed acyclic graph. Each layer of the unrolled RNN model uses the same weights and biases for each unrolled RNN timestamp, although the outputs and the internal states of each unrolled RNN timestamp display different values for each timestamp. Therefore, while outputs and internal states require one copy per timestamp, keeping just one copy of cells weights and biases per layer is possible. This optimization significantly reduces the working set size of RNN workloads, and state-of-the-art deep learning frameworks adopt it like Keras-TensorFlow [35], [36], or PyTorch [37].

Each RNN cell in forward and reverse order input processing produces outputs H_t and $\overleftarrow{H}_t$, respectively. BRNNs combine the output of forward and reverse order cells, which process the same input. BRNNs compute the final output of each cell in forward order by using the following equation:

$$y_t = merge(H_t, \overleftarrow{H}_t) \quad (11)$$

where the $merge$ function combines outputs from forward and reverse order processing cells. This combination can be carried out via operations like summation, multiplication, average, or concatenation. For each layer in deep BRNN models, the cell output of $merge$ (11) is fed to the next layer RNN cell in forward order, and the corresponding cell in reverse order. For example, the $1f3r$ cell in Fig. 1 feeds cells $4f$ and $6r$. Several merge operations combine reverse and forward order cell outputs and propagate this combination to following layer cells.

The most widely used BRNN variants are many-to-one and many-to-many models. Many-to-one BRNN models have the same number of inputs as their sequence length and produce only one output. Such models are widely used for sentiment analysis, recommendation systems, and forecasting

stock market, weather, or traffic [40], among other tasks. Fig. 1 represents a many-to-one BRNN model where the third layer produces a single final output composed of the hidden state outputs of cells $9f$ and $9r$, followed by the $merge$ function defined in Equation (11).

Many-to-many BRNN models generally have several inputs and outputs equivalent to the sequence length, although there are some models for which the number of outputs is not equivalent to the sequence length. Many-to-many BRNNs models are commonly used in tasks as name entity recognition or machine translation [4]. These models have more complex data dependencies in the last layer than many-to-one models because forward-order input processing cells producing the final output must wait for all reverse order cells to produce their corresponding outputs.

State-of-the-art deep learning frameworks, like Keras-TensorFlow [35], [36], or PyTorch [37] apply per-layer barriers between forward and reverse order RNNs. Each layer sequentially performs either forward or reverse order RNNs computations for each timestamp, and then merge the outputs of the forward and reverse order RNNs. It imposes per-layer synchronization points to enforce all merge operations of a specific layer to finish before forward and reverse order cells corresponding to the next layer start processing their input data [17]. While these barrier synchronization points allow the same code to handle BRNN models with different sequence lengths and layer counts, they significantly undermine the parallel performance of BRNN workloads. Indeed, the parallel performance of deep learning frameworks on large core counts is inferior when processing BRNN workloads, as Section IV demonstrates.

III. The B-Par Approach

B-Par (Bidirectional-Parallelization) is a parallel execution model for deep BRNNs. B-Par conceives BRNN forward and backward propagation routines as graphs where nodes represent computation and edges identify data and control dependencies between them. B-Par executes two unidirectional RNN models simultaneously, one model processes input data in forward order, and another in reverse order. B-Par exploits model parallelism on BRNN models by conceiving forward and reverse order input computations in terms of multiple sequential pieces of code, which we denote as *tasks*. A runtime system software orchestrates the parallel execution by considering dependencies between different computing routines and scheduling them across the various computing units of parallel systems.

B-Par relies on the basic structure of deep BRNNs, where a cell on a particular layer depends on a previous cell of the same layer, its counterpart cell of the previous layer, and a cell of the previous layer in reverse order. Orchestrating a BRNN parallel training or inference via task dependencies does not produce any accuracy loss compared to a sequential execution. Like TensorFlow-Keras, B-Par also unrolls BRNNs where weights and biases are shared among the same layer's RNN cells in forward and reverse order input processing cells. Weights and biases are allocated in memory only once per layer since they are shared across all unrolled RNN timestamps in the same layer. Section III-A describes in detail the B-Par approach when applied to many-to-one and many-to-many BRNN models, while Section III-B describes our implementation of B-Par.

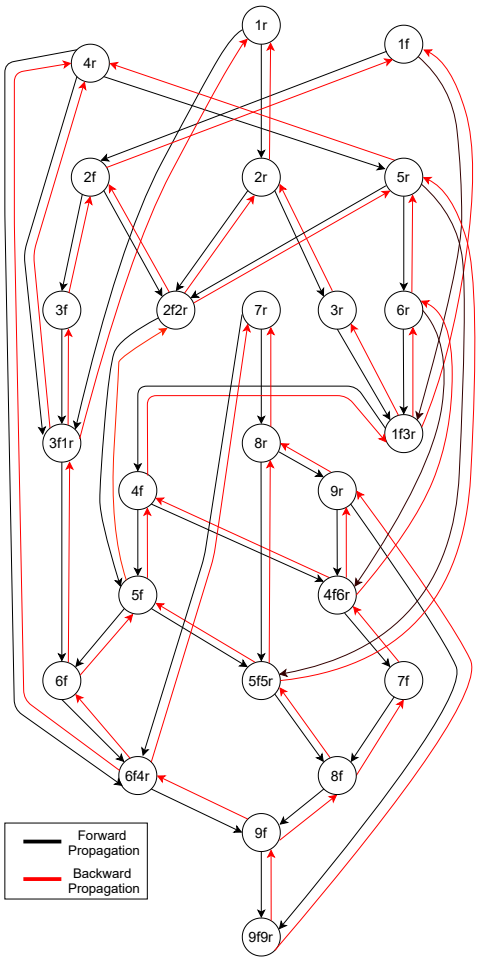

Fig. 2: Many-To-One BRNN model forward and backward propagation dependency graph

A. Applying B-Par to BRNN Models

B-Par maps all computations corresponding to an RNN cell into a single sequential task and orchestrates the parallel run taking into account dependencies across tasks in forward order input processing and reverse order input processing. We consider the BRNN model represented in Fig. 1 to describe the B-Par operation on many-to-one models. In Fig. 1, the forward order cells appear on the left-hand side, while reverse order input processing cells appear on the right-hand side of the figure. Arrows express the data dependencies between RNN cells. B-Par forward propagation for many-to-one BRNNs processes the reverse input processing cells in the sequential order of $1r$, $2r$, and $3r$, and traverses forward order input processing cells in the order of $1f$, $2f$, $3f$. Once the forward and reverse order outputs are computed, B-Par merges them in cells $1f3r$, $2f2r$,

$3f1r$ following Equation (11) and considering operations like sum, subtraction, average, or concatenation. Cells $1f3r$, $2f2r$, $3f1r$ are denoted as merge cells. B-Par repeats the same process for the next layers except the last one. In the last layer of many-to-one BRNNs, B-Par merges the output of the last forward and reverse order processing cells: cells $9f$ and $9r$, in the example we display in Fig. 1. Importantly, B-Par keeps the merge cells as separate tasks to eliminate dependencies between forward and reverse order processing cells belonging to the same layer. This separation permits B-Par to execute forward and reverse order cells in parallel.

Fig. 2 represents the dependency graph concerning the forward and backward propagations of the many-to-one BRNN model illustrated in Fig. 1. Each graph node represents a sequential task that encapsulates all computations corresponding to a single RNN cell. Tasks are created in the topological order during the parallel run representing from top to bottom for the forward propagation and from bottom to top for the backward propagation. Black arrows represent dependencies across the forward propagation, which starts from cells $1r$ and $1f$. Arrows begin in cells producing a piece of data and finish on cells consuming these data. For example, Fig. 2 shows that cell $2r$ has two output dependencies on cells $2f2r$ and $3r$ in the case of forward propagation. Similarly, red arrows appearing in Fig. 2 represent the backward propagation of the many-to-one BRNNs model that appears in Fig. 1. The backward propagation task dependency graph begins from the bottom of the forward propagation dependency graph; that is, it starts from cell $9f9r$ and proceeds in reverse order.

A many-to-many BRNN model is similar to the many-to-one case except for the last layer. Many-to-many models combine the forward and reverse order RNNs cell outputs to produce N outputs, where N is the model's sequence length. Due to the last layer, many-to-many BRNNs models have slightly higher numbers of tasks and complex dependencies than the many-to-one BRNNs model.

B. Implementation of B-Par

Our implementation of B-Par is based on encapsulating sequences of algebraic operations in sequential pieces of work, called tasks, that have data dependencies across them. These algebraic operations correspond to the mathematical transformation required to update the state of RNN cells, and are specified by Equations 1- 10. We use source code annotations to define sequential pieces of work corresponding to single RNN cells updates. These annotations specify all input and output dependencies per task and are supported by the most relevant parallel programming models [21], [22]. B-Par runs on top of a run-time system software that dynamically generates a task dependency graph by exploiting source code annotations. The run-time system maintains a list of ready-to-be executed tasks and assigns them to computing units as soon as they become available.

A practical and general B-Par implementation requires supporting any BRNN model featuring either BLSTM or BGRU cells considering generic input and hidden dimensions, batch sizes, or sequence lengths. For variable sequence length in between batches, B-Par adjust the computation graph dynamically on run-time. Our implementation conceives BRNN models as lists of algebraic operations that update either reverse or forward order cells, which is a general abstraction able to represent any BRNN. It requires abstract data structures to represent and manage these lists of algebraic operations, as well as pragma annotations describing complex and input-specific dependencies.

Algorithm 1 B-Par Main Routines and Parameters

```
1: c_f                          ▷ List of algebraic operations to update all forward order cells
2: c_r                          ▷ List of algebraic operations to update all reverse order cells
3: start_f, end_f               ▷ Array of start and end indices of Forward cells in c_f
4: start_r, end_r               ▷ Array of start and end indices of Reverse cells in c_r
5: action                       ▷ Forward or backward propagation
6:
7: function ForwardorderRNNs − ForwardPropagation(c_f,c_r, start_f, end_f)
8:     Forward Propagation for Forward order RNN cells          ▷ Pseudo-code in Algorithm 2
9:
10: function ReverseorderRNNs − ForwardPropagation(c_r, start_r, end_r)
11:    Forward Propagation for Reverse order RNN cells          ▷ Pseudo-code in Algorithm 3
12:
13: function ForwardorderRNNs − BackwardPropagation(c_f,c_r, start_f, end_f)
14:    Backward Propagation for Forward order RNN cells
15:
16: function ReverseorderRNNs − BackwardPropagation(c_r, start_r, end_r)
17:    Backward Propagation for Reverse order RNN cells
18:
19: function FwdBwdComputations(c,start,end,action)
20:    for node ← start to end do
21:        if c[node]− > Child == TRUE then
22:            algebra_operation[c[node]− > op](c[node], action)
23:
24: function Merge_Task(c_f,e_f,tmp_c_r,action)
25:    #pragma omp task in(c_f[e_f-2], tmp_c_r) out(c_f[e_f])
26:    FwdBwdComputations((e_f-1),e_f, c_f, action)
```

1) Fundamental Data-Structures and Routines of B-Par: B-Par implementation performs forward or backward propagation using the two fundamental data structures: c_f, which contains the list of all necessary algebraic operations to update all forward order cells, and c_r, which does so for reverse order cells. To properly index these lists of algebraic operations, B-Par makes use of the $start\_f$ and end_f arrays, which index the first and the last algebraic operation per each RNN cell in c_f, respectively, and $start\_r$ and end_r, which are their counterpart arrays for c_r. B-Par also considers an $action$ parameter to specify whether it is performing a forward or a backward propagation. These 5 parameters are listed in the first lines of Algorithm 1. Our B-Par implementation also keeps the state of all its RNN cells in a data structure. We omit this data structure in Algorithm 1 for pseudo-code readability and simplicity purposes.

B-Par updates the RNN cells state by using four main routines: Forward propagation for forward cells, forward propagation for reverse cells, backward propagation for forward cells, and backward propagation for reverse cells. Algorithm 1 lists these four routines and their parameters in lines 7, 10, 13, 16.

Algorithm 2 Forward order RNNs - Forward Propagation

```
1: cell_f = −1                                   ▷ Identifier of forward order RNN cells.
2: cell_r = 0                                    ▷ Identifier of reverse order RNN cells.
3: for l ← 0 to #layers do                       ▷ Loop over all layers.
4:     cell_r = (l + 1) ∗ seq.length − 1
5:     for u ← 0 to #seq.length do               ▷ Loop over RNN cells of the same layer.
6:         ++ cell_f
7:         s_f = start_f[cell_f], e_f = end_f[cell_f]   ▷ Start and end indices of cell_f
8:         if u > 0 and l > 0 then
9:             if l == (#layers -1) then
10:                if u == (seq.length -1) then
11:                    #pragma omp task in(c_f[s_f-1], c_f[end_f[cell_f- seq.length]]) out(c_f[e_f-9])
12:                    FwdBwdComputations(s_f, (e_f-9), c_f, FWD)
13:                    #pragma omp task in(c_f[e_f-9], c_r[end_r[#layers × seq.length-1]]) out(c_f[e_f])
14:                    FwdBwdComputations((e_f-8), e_f, c_f, FWD)
15:                else
16:                    #pragma omp task in(c_f[s_f-1], c_f[end_f[cell_f- seq.length]]) out(c_f[e_f])
17:                    FwdBwdComputations(s_f, e_f, c_f, FWD)
18:            else
19:                #pragma omp task in(c_f[s_f-3], c_f[end_f[cell_f- seq.length]]) out(c_f[e_f-2])
20:                FwdBwdComputations(s_f, (e_f-2), c_f, FWD)
21:                Merge_Task(c_f,e_f,c_r[end_r[cell_r], FWD)
22:        else
23:            if u = 0 and l > 0 then
24:                if l == (#layers -1) then
25:                    #pragma omp task in(c_f[end_f[cell_f- seq.length]]) out(c_f[e_f])
26:                    FwdBwdComputations(s_f, e_f, c_f, FWD)
27:                else
28:                    #pragma omp task in(c_f[end_f[cell_f- seq.length]]) out(c_f[e_f-2])
29:                    FwdBwdComputations(s_f, (e_f-2), c_f, FWD)
30:                    Merge_Task(c_f,e_f,c_r[end_r[cell_r], FWD)
31:            else
32:                if u > 0 and l = 0 then
33:                    #pragma omp task in(c_f[s_f-3]) out(c_f[e_f-2])
34:                    FwdBwdComputations(s_f, (e_f-2), c_f, FWD)
35:                    Merge_Task(c_f,e_f,c_r[end_r[cell_r], FWD)
36:                else
37:                    if u = 0 and l = 0 then
38:                        #pragma omp task in(c_f[s_f]) out(c_f[e_f-2])
39:                        FwdBwdComputations(s_f, (e_f-2), c_f, FWD)
40:                        Merge_Task(c_f,e_f,c_r[end_r[cell_r], FWD)
41:    −− cell_r
```

Algorithm 3 Reverse order RNNs - Forward Propagation

```
1: cell_f = 0                                    ▷ Identifier of forward order RNN cells.
2: cell_r = −1                                   ▷ Identifier of reverse order RNN cells.
3: for l ← 0 to #layers do                       ▷ Loop over all the layers.
4:     cell_f = l ∗ seq.length − 1
5:     for u ← 0 to #seq.length do               ▷ Loop over RNN cells of the same layer.
6:         ++ cell_r
7:         s_r = start_r[cell_r], e_r = end_r[cell_r]   ▷ Start and end indices of cell_r
8:         if u > 0 and l > 0 then
9:             #pragma omp task in(c_r[s_r-1], c_f[end_f[cell_f]]) out(c_r[e_r])
10:            FwdBwdComputations(s_r, e_r, c_r, FWD)
11:            −− cell_f
12:        else
13:            if u= 0 and l> 0 then
14:                #pragma omp task in(c_f[end_f[cell_f]]) out(c_r[e_r])
15:                FwdBwdComputations(s_r, e_r, c_r, FWD)
16:                −− cell_f
17:            else
18:                if u > 0 and l = 0 then
19:                    #pragma omp task in(c_r[s_r-1]) out(c_r[e_r])
20:                    FwdBwdComputations(s_r, e_r, c_r, FWD)
21:                else
22:                    if u = 0 and l = 0 then
23:                        #pragma omp task in(c_r[s_r]) out(c_r[e_r])
24:                        FwdBwdComputations(s_r, e_r, c_r, FWD)
```

The two routines to perform forward propagation are described in Section III-B2, and Algorithms 2 and 3 display their pseudo-code. We do not show the pseudo-code of backward propagation routines since it is very close to their forward counterparts. The four main routines for forward and backward propagation make extensive use of the $FwdBwdComputations$ function, which appears in line 19 of Algorithm 1, and it is the fundamental kernel of B-Par. The $FwdBwdComputations$ input parameters are the c data-structure, which is an instantiation of either c_r or c_f; the $start$ and end indices, which index the initial and the final arithmetic operations of the RNN cell being processed; and the $action$ parameter, which describes whether the propagation is either forward or backward. All computations from $start$ to end constitute a sequential task.

The $FwdBwdComputations$ routine goes over all the algebraic operations required to update the RNN cell (line 20 of Algorithm 1). The $algebra\_operation$ carries out a specific operation defined by $c[operation] -> op$, which is where the c data structure holds the corresponding operations to be performed. Equations 1- 10 define these algebra operations depending on the particular RNN variant we are updating (line 22). Algorithm 1 also defines the $Merge\_Task$ function, which corresponds to merge tasks. This routine relies on the $FwdBwdComputations$ but adds an additional pragma annotation containing specific dependencies of merge tasks.

2) Routines to Compute Forward Propagation for Reverse and Forward Order RNN Cells: For many-to-one BRNNs, Algorithm 2 displays the high-level pseudo-code representation of forward order RNNs for the case of forward propagation. Forward order tasks are instantiated by different pragma annotations, which specify the input and output dependencies of the corresponding RNN cell. In forward order input processing, B-Par handles the cells responsible for merging the output of forward and reverse order input computations by calling the $Merge\_Task$ routine, defined in line 24 of Algorithm 1. Every merge cell has input dependencies on both reverse and forward order RNN cells, while the other forward order cells have input dependencies only on forward order cells.

Algorithm 2 is composed of two main loops that traverse all network layers and all RNN cells of each layer. We represent these two loops in lines 3 and 5. Algorithm 2 implements a complex heuristic depending on the layer and the cell's relative position within each layer, which variables l and u describe. Depending on the u and l variables, Algorithm 2 defines different input and output dependencies before calling the $FwdBwdComputations$ routine, which performs the algebraic transformations required to update the state of the RNN cell that is being processed. Similarly, the $Merge\_Task$ has different dependencies depending on the specific layer and cell we are updating.

Algorithm 3 contains the pseudo-code of the forward propagation routine of reverse order cells. This pseudo-code has a very similar structure as Algorithm 2 since it also traverses all the layers of the model and all the cells of each layer. Algorithm 3 is simpler than Algorithm 2 since it does not

deal with merge tasks. Forward propagation of reverse order cells also requires the definition of all input and output dependencies via pragma annotations. Algorithms 2 and 3 interact with each other such that the run-time system launches layer-wise tasks consisting of forward and reverse order and merge tasks, so on for the successive layers of BRNNs. We do not provide the pseudo-code of the backward propagation routines as it is very similar to their forward counterparts.

We specify input and output dependencies in terms of pragma annotations, making it possible for the run-time system software to dynamically represent the parallel workload via a task dependency graph, where each task represents the update of a single RNN cell. This parallel execution model simplifies our implementation, increases its flexibility, and does not impose any barrier synchronization point between network layers. Our B-Par implementation can be trivially extended to support data-parallelism by dividing a batch into multiple mini-batches and processing each one of them in parallel. B-Par combines the contribution of each mini-batch before computing the weight updates. This is expressed at the source code level by adding dependencies to tasks that compute gradient updates. These dependencies enforce gradient synchronization among model replicas.

IV. Evaluation

This section evaluates the performance of B-Par against the primary state-of-the-art deep learning frameworks supporting BRNNs. It compares B-Par performance on many-core CPU devices against state-of-the-art approaches running on either many-core CPUs or GPUs, and it shows how the exploitation of model parallelism that B-Par achieves without barrier points provides very significant performance improvements.

A. Experimental Setup

Our experiments are performed on a dual-socket 48-core Intel Xeon system and an Nvidia Tesla V100 GPU. Table I summarizes the main aspects of our two experimental platforms. The Intel CPU cache storage is as follows: L1D 32K; L1I 32K; L2 cache 1024K; L3 cache 33792K. The L3 cache is shared among the cores belonging to the same socket. On the CPU system, we use the Intel optimized TensorFlow [41] installation for our experiments and a PyTorch [42] installation with all the Intel recommended optimizations. Both TensorFlow-Keras and PyTorch support AVX512, MKL-Parallel, and oneDNN which is an open-source, cross-platform high-performance library of basic building blocks for deep learning applications. Our experiments on the GPU system consider versions of TensorFlow-Keras [35] and PyTorch [37] that leverage cuDNN, a GPU accelerated deep learning library. We show in Table II the specific software versions and related libraries of our experimental setup.

TABLE I: Hardware Experimental Platforms

Computational Unit	#CPUs	Memory	OS
2 Intel XEON Platinum 8160 @2.1 GHz	2×24	96 GB	SuSe Linux 12 SP2
Tesla V100 SXM2	40	16 GB	Red Hat Enterprise Linux Server 7.5

TABLE II: Software Setup

Framework	Version	Python	MKL	oneDNN	GCC	CUDA	CuDNN
Keras-TensorFlow	2.3.0	3.7.4	2019.04	1.6.0	8.1.0	10.2	8.0.3
PyTorch	1.7.1	3.7.4	2020.0.0	1.6.0	8.1.0	10.2	8.0.3
B-Par	-	-	2019.4	-	8.1.0	-	-

TensorFlow-Keras: To achieve the best performance for TensorFlow-Keras and PyTorch, we perform 64 experiments for each model configuration with a combination of inter and intra-threads from 1, 2, 4, 8, 16, 24, 32, 48. For oneDNN (earlier MKL-DNN), the number of MKL threads is always equal to intra-threads. Previous work [43] discusses the best possible values for inter-, intra-, and MKL-threads for TensorFlow-Keras by exhaustively sweeping the design space, although they do not consider RNN or BRNN models. Settings from Intel produce better performance for RNN and BRNN models than these previous approaches and match our exhaustive exploration [43]. When reporting execution time on 24 or fewer cores, we restrict TensorFlow-Keras execution to a single socket to avoid Non-Uniform Memory Access (NUMA) effects. For TensorFlow-Keras, we always unroll BRNNs to reduce the memory requirements of the training process as described in Section II.

B-Seq: Besides TensorFlow-Keras, PyTorch, and B-Par, our experiments consider a BRNN implementation that entirely relies on data parallelism, *B-Seq*. It splits batches into mini-batches that are processed in parallel. B-Seq only relies on data parallelism and processes each mini-batch sequentially. In contrast, B-Par relies on both data- and model-parallelism, which means that it can split a batch into several mini-batches and process each mini-batch in parallel. Both B-Seq and B-Par use the same OmpSs runtime and unroll BRNNs.

TensorFlow-Keras, PyTorch, B-Seq, and B-Par experiments use the same BRNN initial configuration. All experiments use one thread per core. The OmpSs run-time system [22] is in charge of dynamically scheduling the parallel tasks of B-Par to the compute units. B-Par uses a breadth-first task scheduler with a single global-ready queue. Breadth-first implements a locality-aware mechanism that schedules tasks taking into account their data dependencies. This mechanism schedules a task to run on the same core as a predecessor if the task accesses a piece of data that was already read or written by the predecessor. We study the impact of this locality-aware mechanism by comparing its performance in terms of Instructions per Cycle (IPC), cache Misses per Kilo Instruction (MPKI), and execution time with a task scheduler that does not consider data locality. The ready queue is ordered following a FIFO (First In, First Out) algorithm. B-Par is mapped to MKL-Sequential, while TensorFlow-Keras and PyTorch use MKL-Parallel and oneDNN.

Data-sets: We consider two different data-sets in our evaluation: The TIDIGITS speech corpus data-set [33] and the real-world text-corpus Wikipedia data-set [34]. TIDIGITS contains speech, originally designed and collected to evaluate algorithms for speaker-independent recognition of connected digit sequences. Wikipedia is a data-set of 1.4 billion characters.

Many-to-one BRNNs deal with TIDIGITS, and many-to-many BRNNs process the Wikipedia data-set to perform the next character prediction problem.

B. Performance on Speech Recognition Task

We evaluate B-Par against TensorFlow-Keras and PyTorch running on CPUs and GPUs, and B-Seq on CPUs, considering a speech recognition task on the TIDIGITS data-set covering a wide range of BRNN model sizes. We also evaluate in detail the impact on the performance of model hyper-parameters.

Speed-up on CPUs: We compare single batch training times of B-Par running on CPUs with TensorFlow-Keras and PyTorch on CPUs and GPUs, and B-Seq on CPUs. Tables III and IV report these results measured on 6-layer deep BRNNs composed of LSTMs and GRUs, respectively. B-Par is always faster than TensorFlow-Keras, PyTorch on CPUs for all the different configurations. Training time includes the forward and backward propagation plus the gradient update time per batch. The first four columns present the considered model configurations in terms of input dimension, hidden size, batch size, sequence length, which define the number of trainable parameters displayed in the fifth column.

B-Par significantly outperforms TensorFlow-Keras on CPUs with a speed-up within the range of $1.50\times$ (Table III configuration: 256/256/1/10) to $2.34\times$ (Table IV configuration: 256/256/1/100), and PyTorch on CPUs with speed-ups from $1.30\times$ (Table III configuration: 256/256/1/2) to $9.16\times$ (Table III configuration: 256/1024/256/100). Large sequence length and hidden size values allow B-Par to expose considerably parallel workloads to the architecture and achieve significant performance speed-ups.

Speed-up on GPUs: On GPUs, B-Par achieves speed-up improvements up to $5.05\times$. B-Par is always faster than TensorFlow-Keras and PyTorch when both batch size and sequence length are smaller than 10. As the sequence length increases, GPUs become more efficient since BRNN computations involve large matrix-matrix multiplications. PyTorch executions on the GPU system become very inefficient when there are more than 90Million model parameters. We leave the corresponding entries of Tables III, and IV empty since the executions of PyTorch often hung in these scenarios.

Next, we look into the five most relevant parameters that impact the performance of BRNNs training and inference time: hidden size, batch size, number of cores, data parallelism (mini-batch size), and number of layers. We conduct the subsequent experiments considering both 8-layer and 12-layer deep BLSTM models keeping the sequence length set to 100 and input size to 256 unless explicitly stated otherwise. We also demonstrate the potential of B-Par in terms of exploiting data locality and reuse opportunities when a cache locality-aware scheduler is applied. Finally, we show the overhead B-Par incurs in task creation, scheduling, and synchronization are negligible.

Varying number of cores and mini-batch size training: Fig. 3 shows the scalability of B-Par on 8- and 12-layer BLSTM models while using different mini-batch sizes and

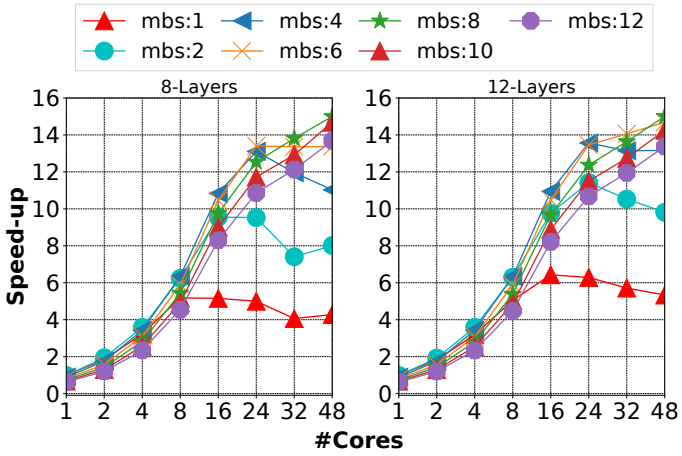

Fig. 3: B-Par speed-up against B-Par-mbs:1 on different CPU core counts.

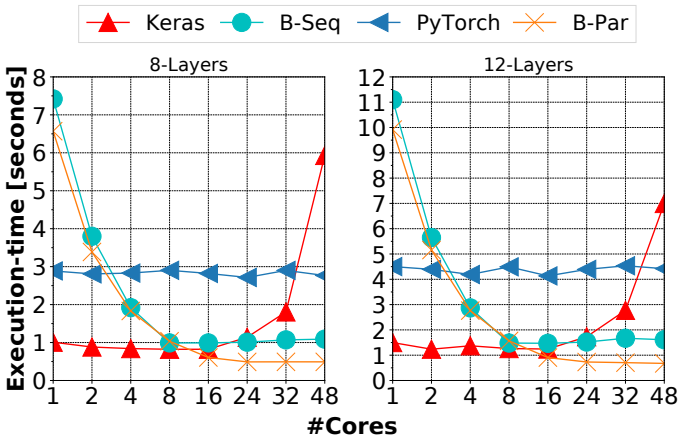

Fig. 4: Keras, B-Seq, PyTorch, and B-Par batch training execution time on different CPU core counts.

cores against single core execution-time of B-Par using a mini-batch size of 1. mbs:N signifies that batch size is divided among N batches that run in parallel. B-Par is evaluated considering mini-batch sizes of 1, 2, 4, 6, 8, 10, 12 on different core counts. The best speed-up is seen on mbs:8 while using all the 48 cores as this configuration exposes ample amounts of parallelism to the underlying parallel hardware. Configurations mbs:10 and mbs:12 show sightly worse performance than mbs:8 since they incur more task creation overhead than the mbs:8 configuration.

B-Par configurations mbs:1, mbs:2, and mbs:4 display a performance degradation when run on 32 and 48 cores. NUMA effects appear when our parallel experiments use the two sockets of our experimental platform. There is no NUMA effect when less than the 24 cores of a single socket are used. B-Par configurations mbs:8 mbs:10, and mbs:12 improve their performance when increasing the number of CPUs from 24 to 32 due to the large concurrency of these configurations, which take advantage of the additional cores and provide additional performance despite NUMA effects.

TABLE III: BLSTMs Training times and speed-up of B-Par on CPU against Keras(K), PyTorch(P) on CPU and GPU systems.

Model Configurations					BLSTMs batch execution time (ms)						Speed-up of B-Par-CPU wrt			
Input Size	Hidden Size	Batch Size	Seq Len	Parameters	K-CPU	K-GPU	P-CPU	P-GPU	BSeq-CPU	BPar-CPU	K-CPU	K-GPU	P-CPU	P-GPU
64	256	128	100	5.9M	1,770.76	123.79	3,215.68	590.5726	2,364.00	989.06	1.79	0.13	3.25	0.60
256	256	128	100	6.3M	1,770.15	132.67	3,956.06	590.2132	2,419.80	932.55	1.90	0.14	4.24	0.63
1024	256	128	100	7.8M	1,816.53	193.36	3,663.28	595.0635	2,726.55	1,149.55	1.58	0.17	3.19	0.52
256	256	1	2	6.3M	17.47	24.52	20.51	24.0476	20.21	14.94	1.17	1.64	1.37	1.61
256	256	1	10	6.3M	37.29	29.27	54.70	64.6389	60.76	24.80	1.50	1.18	2.21	2.61
256	256	1	100	6.3M	276.68	80.71	461.45	515.6153	439.25	143.21	1.93	0.56	3.22	3.60
64	256	256	100	5.9M	2,751.70	177.08	5,240.83	562.2858	4,262.18	1,566.60	1.76	0.11	3.35	0.36
64	1024	256	100	92.8M	28,489.52	1,276.98	147,839.40	-	71,038.30	17,378.61	1.64	0.07	8.51	-
256	256	256	100	6.3M	2,770.82	201.12	5,412.32	559.3186	4,352.02	1,581.97	1.75	0.13	3.42	0.35
256	1024	256	100	94.4M	28,571.33	1,316.64	143,332.02	-	71,715.42	15,640.74	1.83	0.08	9.16	-
1024	256	256	100	7.8M	2,893.43	303.52	5,713.00	558.8649	4,546.46	1,830.35	1.58	0.17	3.12	0.31
1024	1024	256	100	100.7M	28,721.38	1,497.25	117,934.39	-	71,521.05	16,143.40	1.78	0.09	7.31	-

TABLE IV: BGRUs Training times and speed-up of B-Par on CPU against Keras(K) and PyTorch(P) on CPU and GPU systems.

Model Configurations					BGRUs batch execution time (ms)						Speed-up of BPar-CPU wrt			
Input Size	Hidden Size	Batch Size	Seq Len	Parameters	K-CPU	K-GPU	P-CPU	P-GPU	BSeq-CPU	BPar-CPU	K-CPU	K-GPU	P-CPU	P-GPU
64	256	128	100	4.4M	1,246.98	125.36	2,726.72	604.0995	1,702.27	690.83	1.81	0.18	3.95	0.87
256	256	128	100	4.7M	1,254.30	153.45	2,303.21	605.8498	1,746.60	729.82	1.72	0.21	3.16	0.83
1024	256	128	100	5.9M	1,333.97	189.25	6,415.08	608.0237	1,950.52	856.44	1.56	0.22	7.49	0.71
256	256	1	2	4.7M	16.05	23.66	22.03	22.8991	12.77	9.43	1.70	2.51	2.34	2.43
256	256	1	10	4.7M	34.23	28.83	59.74	65.5189	39.12	18.39	1.86	1.57	3.25	3.56
256	256	1	100	4.7M	246.11	66.31	504.54	531.1096	313.68	105.17	2.34	0.63	4.80	5.05
64	256	256	100	4.4M	2,239.56	144.54	3,035.85	639.5767	3,060.31	1,160.42	1.93	0.12	2.62	0.55
64	1024	256	100	69.6M	26,210.06	986.15	32,303.64	-	42,322.66	15,020.14	1.74	0.07	2.15	-
256	256	256	100	4.7M	2,256.72	166.10	3,207.68	638.7488	3,120.84	1,277.92	1.77	0.13	2.51	0.50
256	1024	256	100	70.8M	26,111.23	1,019.34	50,828.08	-	41,752.00	13,156.51	1.98	0.08	3.86	-
1024	256	256	100	5.9M	2,359.49	292.00	6,118.97	635.2685	3,310.15	1,417.83	1.66	0.21	4.32	0.45
1024	1024	256	100	75.5M	26,253.30	1,157.89	41,555.13	-	43,156.39	13,741.52	1.91	0.08	3.02	-

Fig. 4 compares the performance of Keras, B-Seq mbs:8, PyTorch and B-Par mbs:8, when we consider core counts 1, 2, 4, 8, 16, 24, 32, and 48. Since B-Seq just exploits data parallelism, it cannot expose more than 8 parallel software components to the multi-core architecture; hence B-Seq achieves best execution time of 0.89 seconds when run on 8 cores. Increasing core count does not provide any benefit for B-Seq. B-Par delivers much better performance than B-Seq due to model-parallelism. The performance of B-Seq is similar to Keras on 8 and 16 cores. However, Keras suffers from NUMA effects when running on dual-socket configurations on large core counts. As we consider core counts larger than 16, B-Par significantly outperforms TensorFlow-Keras and PyTorch. Best execution time of B-Par is 0.44 seconds on 48 cores.

Varying hidden/batch size: Fig. 5 reports single batch training time of B-Par, Keras-CPU, PyTorch-CPU, and B-Seq for the 8-layer and 12-layer BLSTM models on batch sizes varying from 128 to 1024, and a combination of 128 and 256 hidden states. We run these experiments considering core counts 1, 2, 4, 8, 16, 24, 32, and 48. We report the best execution time out of all core counts. B-Par consistently outperforms Keras-CPU and PyTorch-CPU and achieves very significant performance speed-ups within the $1.58 - 6.40\times$ range for all configurations. PyTorch performs worse among all configurations.

Varying number of layers: Fig. 6 displays the single batch execution time of training and inference for B-Par, Keras-CPU, PyTorch-CPU, and B-Seq while varying the number of

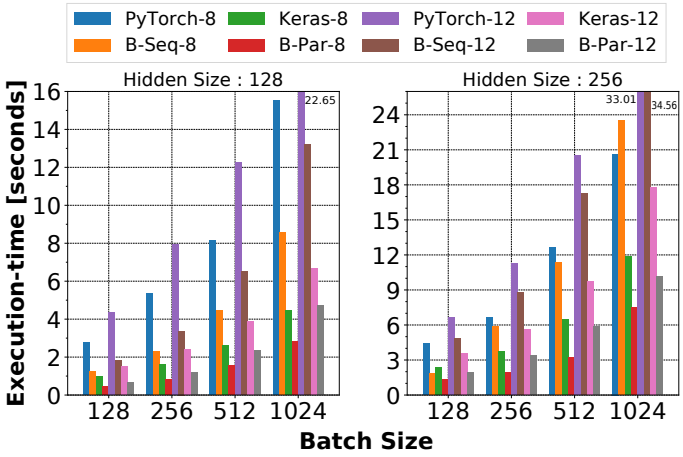

Fig. 5: B-Par, Keras, PyTorch and B-Seq on CPUs with different batch and hidden state sizes.

layers. B-Par performs and scales the best with an increasing number of layers for both training and inference time due to the significant parallelism it exposes to the hardware. For a 12-layer BLSTMs model, the speed-up achieved by B-Par is $5.89\times$ and $6.40\times$ for inference and training, respectively. B-Par does not impose per-layer barriers, which explains its superior performance with respect to Keras and PyTorch, particularly for the 12-layer model.

Cache locality impact: We evaluate the impact of the locality-aware mechanism we describe in in Section IV-A.

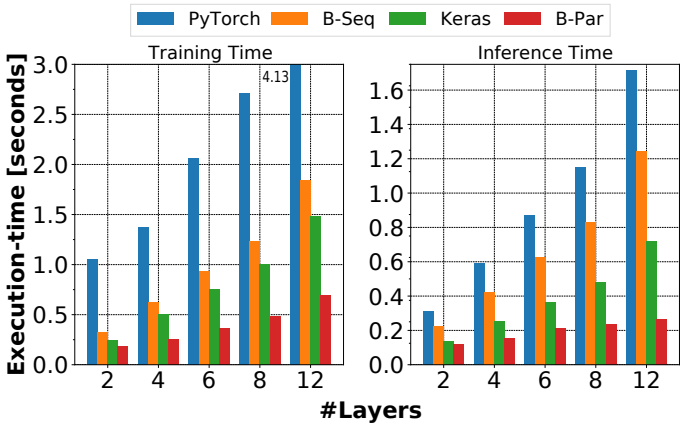

Fig. 6: B-Par, B-Seq, Keras and PyTorch on CPUs with different layer counts.

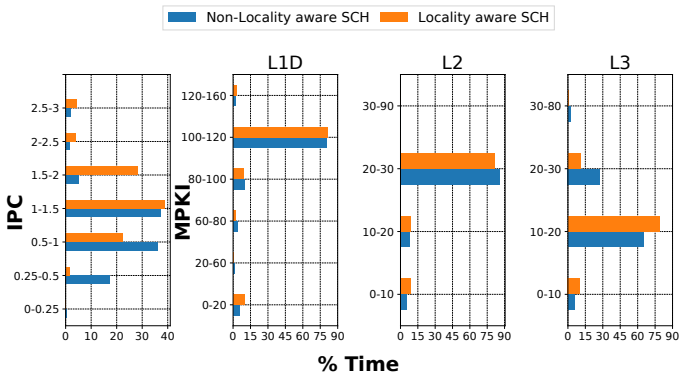

Fig. 7: Impact of locality awareness on an 8-layer BLSTM.

We consider an 8-layer BLSTMs model with 31.7Million parameters that does not fit in the cache hierarchy of the CPU system we describe in Section IV-A. We show our performance measurements in Fig. 7. Locality-aware scheduling increases the percentage of training time at which the execution delivers an IPC ratio within the 1.5-2 interval from 5% to 29%, as the left-hand-side histogram of Fig. 7 shows. A significant L3 MPKI reduction explains this improvement. Indeed, the right-hand side histogram of Fig. 7 displays a drop from 28% to 10% of the execution time portion where 20-30 L3 MPKI ratios are observed. Finally, we observe that locality-aware scheduling reduces average training batch time by 20% with respect to locality-oblivious approaches.

Task-granularity: We measure the task granularity of B-Par considering a BLSTM model with hyper-parameters Seq Length=100, Batch Size=128, Input Size=64, and Hidden Size=512. B-Par triggers a total of 368,240 tasks in this scenario. The average working set size of tasks processing a single LSTM cell is 4.71 MB, which does not fit the cache hierarchy of the CPU system described in Section IV-A. Merge tasks corresponding to Equation (11) have a much smaller working set than LSTM tasks. Task granularity ranges from 272.8 to 315,178.31 microseconds, while the average duration is 13,052.23 microseconds. This granularity keeps the B-Par overhead in terms of task creation, scheduling, and synchronization ten times smaller than the total time spent in actually running parallel tasks containing user-level code.

Memory Consumption: The working set size of B-Par when considering an 8-layer-BLSTM model at mbs:6 without per per-layer synchronization is 75.36 MB, while adding per-layer synchronization reduces this size to 28.26 MB. This difference is explained by the average number of tasks B-Par processes in parallel when per-layer synchronization is removed, 16, which is much larger than the average number of active tasks when per-layer synchronization is enabled, 6. The additional parallelism enabled by removing per-layer synchronization increases the working set size of B-Par parallel workloads while delivering large performance gains concerning approaches requiring per-layer synchronization. However, there is no data and accuracy loss between with and without per-layer synchronization of B-Par execution.

C. Performance on Next Character Prediction

We consider the next character prediction problem on a real-world Wikipedia text-corpus, where a many-to-many BRNN model is used. We consider B-Par and Keras in this section. We omit PyTorch since Tables III, IV indicate that Keras is much faster than PyTorch on CPUs. Fig. 8 reports the single batch training-time of B-Par and Keras-CPU for BLSTMs and BGRUs. We vary the batch size and hidden size parameters. B-Par achieves a maximum speed-up of $1.54\times$, $2.17\times$, $2.38\times$, and $2.44\times$ for 2, 4, 8, and 12 layers, respectively, with respect to Keras across all the configurations that Fig. 8 displays.

V. Related Work

BRNNs are typically accelerated using the same techniques as basic RNN models [44], [45]. Techniques like pruning, pipelining, batching, and choosing optimal data layouts are standard techniques to accelerate both RNNs and BRNNs models [46]. B-Par can be applied on top of these techniques since it is an orthogonal approach. GPUs handle RNN workloads [47]–[49] via extensive exploitation of data parallelism. B-Par is the first approach relying on both data and model parallelism to accelerate BRNNs without imposing costly barrier synchronization points.

ParaX [17] boosts the performance of deep learning on many-core CPUs by effectively alleviating bandwidth contention. While B-Par and ParaX are built on the same observation, that is, the poor performance of deep learning frameworks due to per-layer synchronization points, ParaX focuses on the memory bandwidth contention issue brought by the fact that all threads leave the barrier synchronization point at the same time. Instead, B-Par removes all barrier points and exposes all possible sources of parallelism, either data or model parallelism, to the parallel hardware.

Some previous approaches characterize RNNs workloads and identify low data reuse as one of the main factors causing low performance [50]. Section IV-B shows how the dynamic scheduling approach of B-Par successfully exploits cache locality and elegantly manages one of the leading performance

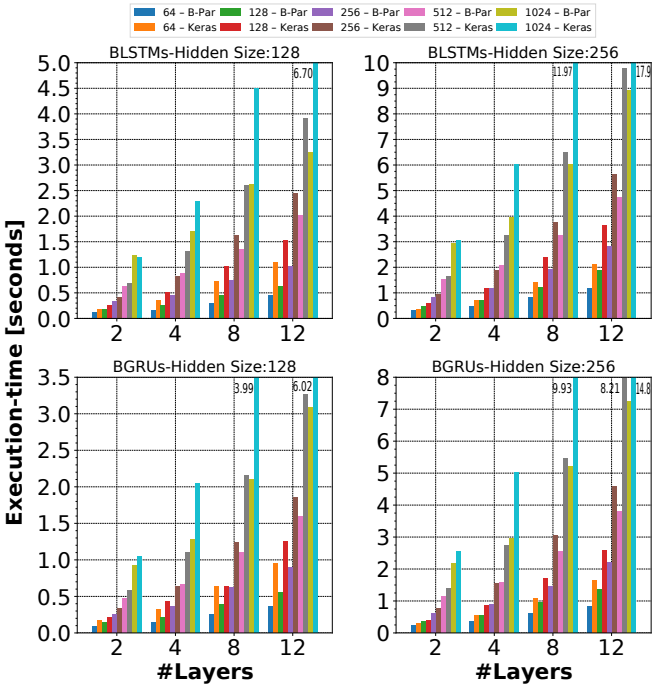

Fig. 8: Batch training execution time of B-Par and Keras for the Next Character Prediction task on a multi-core CPUs device.

issues of RNN workloads. As such, B-Par maximizes the parallelism of BRNN workloads and exploites opportunities for data reuse at the cache hierarchy level.

Recent work on RNNs acceleration primarily targets performance improvements via compiler or run-time system optimizations [51] or scheduling computation graphs [52] only for unidirectional RNNs. Chen et al. [53] perform compiler optimization for RNNs and variants which only support unidirectional RNNs. Efforts are being made by popular deep learning frameworks like the PyTorch-1.6.0 beta release to exploit model parallelism via simple fork-join parallel constructs [37]. Compiler approaches to accelerate DNNs do not support bidirectional RNNs [53], and are orthogonal to B-Par.

There is scope for attention and transformer models combined with BRNNs [10], [12], [54]. In Ezen-Can et al. work [16], BLSTMs outperform transformer-based BERT models for small corpus. Techniques such as reducing the number of parameters in GRUs [55] and using momentum with RNNs to handle the vanishing gradient problem [56] can easily be leveraged by B-Par. CPU-based back-propagation is shown to be significantly faster than GPUs in training models composed of hundreds of millions of parameters [57]. B-Par can accelerate this process. Moreover, B-Par can accelerate BRNNs when combined with CNNs for sentiment analysis [58], and BRNNs with attention models for Chatbots [11].

VI. Conclusion

This paper demonstrates that B-Par delivers good scalability for the training and inference phases of BRNN models on multi-core CPU devices. B-Par outperforms state-of-the-art TensorFlow-Keras and PyTorch frameworks since it does not impose barrier synchronization points between layers. We provide experimental evidence on the benefits of using B-Par on relatively large core counts by achieving up to $2.34\times$ and $9.16\times$ speed-ups concerning TensorFlow-Keras and PyTorch, respectively. We also show strong scalability experimental results concerning B-Par. This paper shows that BRNNs model training and inference can achieve excellent performance on multi-core CPUs, which complements the state-of-the-art approaches based on executing BRNNs models on GPUs. The B-Par's task-graph execution model could be easily applied to a wide range of deep learning models, including transformers and attention mechanisms.

Acknowledgment

This work is partially supported by the Generalitat de Catalunya (contract 2017-SGR-1414) and the Spanish Ministry of Science and Technology through the PID2019- 107255GB project. Marc Casas has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under the Ramon y Cajal fellowship No. RYC-2017-23269.

References

[1] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[2] M. Hermans and B. Schrauwen, "Training and analyzing deep recurrent neural networks," *Advances in Neural Information Processing Systems*, 01 2013.

[3] K. Chen and Q. Huo, "Training deep bidirectional lstm acoustic model for lvcsr by a context-sensitive-chunk bptt approach," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 7, pp. 1185–1193, 2016.

[4] M. Sundermeyer and T. Alkhouli, "Translation modeling with bidirectional recurrent neural networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Oct. 2014. [Online]. Available: https://www.aclweb.org/anthology/D14-1003

[5] M. Liwicki, A. Graves, S. Fernàndez, H. Bunke, and J. Schmidhuber, "A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks," in *ICDAR 2007*, 2007.

[6] A. N. Jagannatha and H. Yu, "Bidirectional rnn for medical event detection in electronic health records," in *Association for Computational Linguistics*. NIH Public Access, 2016, p. 473.

[7] M. Chen, G. Ding, S. Zhao, H. Chen, Q. Liu, and J. Han, "Reference based lstm for image captioning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, no. 1, 2017.

[8] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, vol. 20, no. 5, p. 14, 2015.

[9] V. Passricha and R. K. Aggarwal, "A hybrid of deep cnn and bidirectional lstm for automatic speech recognition," *Journal of Intelligent Systems*, vol. 1, no. ahead-of-print, 2019.

[10] Z. Tüske, G. Saon, K. Audhkhasi, and B. Kingsbury, "Single headed attention based sequence-to-sequence model for state-of-the-art results on switchboard-300," *arXiv preprint arXiv:2001.07263*, 2020.

[11] M. Dhyani and R. Kumar, "An intelligent chatbot using deep learning with bidirectional rnn and attention model," *Materials Today: Proceedings*, vol. 34, pp. 817–824, 2021.

[12] Z. Huang, P. Xu, D. Liang, A. Mishra, and B. Xiang, "Trans-blstm: Transformer with bidirectional lstm for language understanding," *arXiv preprint arXiv:2003.07000*, 2020.

[13] C. Ronran and S. Lee, "Effect of character and word features in bidirectional lstm-crf for ner," in *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 2020, pp. 613–616.

[14] D. Z. Liu and G. Singh, "A recurrent neural network based recommendation system."

[15] S. Mukherjee and A. H. Awadallah, "Xtremedistil: Multi-stage distillation for massive multilingual models," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020.

[16] A. Ezen-Can, "A comparison of lstm and bert for small corpus," *arXiv preprint arXiv:2009.05451*, 2020.

[17] L. Yin and Zhang, "Parax: Boosting deep learning for big data analytics on many-core cpus," *Proc. VLDB Endow.*, vol. 14, no. 6, 2021.

[18] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural networks*, vol. 18, no. 5-6, pp. 602–610, 2005.

[19] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 12 2014.

[20] L. Tang, Y. Wang, T. L. Willke, and K. Li, "Scheduling computation graphs of deep learning models on manycore cpus," 2018.

[21] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, pp. 46 – 55, 02 1998.

[22] A. Duran, E. Ayguadé, R. M. Badia, and J. Labarta, "Ompss: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, pp. 173–193, 06 2011.

[23] D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryaev, and A. Heinecke, "Optimizing deep learning recommender systems training on cpu cluster architectures," pp. 1–15, 2020.

[24] S. Mittal, P. Rajput, and S. Subramoney, "A survey of deep learning on cpus: opportunities and co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[25] "Fugaku Supercomputer," https://postk-web.r-ccs.riken.jp/, 2022.

[26] "Top 500 List," https://top500.org/lists/top500/2021/11/, 2021.

[27] "Summit Supercomputer," https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/, 2022.

[28] H. et al., "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[29] U. Gupta, S. Hsia, and Saraph, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 982–995.

[30] M. Naumov, J. Kim, and Mudigere, "Deep learning training in facebook data centers: Design of scale-up and scale-out systems," *arXiv preprint arXiv:2003.09518*, 2020.

[31] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[32] P. Blacker, C. P. Bridges, and S. Hadfield, "Rapid prototyping of deep learning models on radiation hardened cpus," in *2019 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2019, pp. 25–32.

[33] R. G. Leonard and G. Doddington, "Tidigits speech corpus," *Texas Instruments, Inc*, 1993.

[34] M. Hermans and B. Schrauwen, "Training and analyzing deep recurrent neural networks," *Advances in Neural Information Processing Systems*, 01 2013.

[35] F. Chollet, "keras," https://github.com/keras-team/keras, 2015.

[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, and I. G. et. al, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[37] A. Paszke, S. Gross, F. Massa, and Lerer, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[38] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[39] R. K. Sharma and M. Casas, "Wavefront parallelization of recurrent neural networks on multi-core architectures," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20, New York, NY, USA, 2020.

[40] K. A. Althelaya and El-Alfy, "Stock market forecast using multivariate analysis with bidirectional and stacked (lstm, gru)," in *2018 21st Saudi Computer Society National Computer Conference (NCC)*. IEEE.

[41] "Using the Intel optimized tensroflow," https://intel.ly/2RPWklu, 2018.

[42] "Using the Intel optimized Pytorch," https://github.com/intel/intel-extension-for-pytorch, 2020.

[43] Y. E. Wang, C.-J. Wu, X. Wang, K. Hazelwood, and D. Brooks, "Exploiting parallelism opportunities with deep learning frameworks," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, Dec. 2021. [Online]. Available: https://doi.org/10.1145/3431388

[44] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee, "Recent advances in recurrent neural networks," 12 2017.

[45] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.

[46] S. Mittal and S. Umesh, "A survey on hardware accelerators and optimization techniques for rnns," *Journal of Systems Architecture*, p. 101839, 2020.

[47] T. Lei, Y. Zhang, S. Wang, H. Dai, and Y. Artzi, "Simple recurrent units for highly parallelizable recurrence," 01 2018, pp. 4470–4481.

[48] J. Appleyard, T. Kociský, and P. Blunsom, "Optimizing performance of recurrent neural networks on gpus," *CoRR*, vol. abs/1604.01946, 2016. [Online]. Available: http://arxiv.org/abs/1604.01946

[49] C. Holmes, D. Mawhirter, Y. He, F. Yan, and B. Wu, "Grnn: Low-latency and scalable rnn inference on gpus," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–16.

[50] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," 07 2018.

[51] R. Baghdadi and Ray, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *2019 IEEE/ACM CGO*. IEEE, 2019, pp. 193–205.

[52] L. Tang, Y. Wang, T. L. Willke, and K. Li, "Scheduling computation graphs of deep learning models on manycore cpus," *CoRR*, vol. abs/1807.09667, 2018. [Online]. Available: http://arxiv.org/abs/1807.09667

[53] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," 2018.

[54] Y. Wang, X. Zhang, M. Lu, H. Wang, and Y. Choe, "Attention augmentation with multi-residual in bidirectional lstm," *Neurocomputing*, vol. 385, pp. 340–347, 2020.

[55] M. O. Turkoglu, S. D'Aronco, J. D. Wegner, and K. Schindler, "Gating revisited: Deep multi-layer rnns that can be trained," *arXiv preprint arXiv:1911.11033*, 2019.

[56] T. M. Nguyen, R. G. Baraniuk, A. L. Bertozzi, S. J. Osher, and B. Wang, "Momentumrnn: Integrating momentum into recurrent neural networks," *arXiv preprint arXiv:2006.06919*, 2020.

[57] S. Daghaghi, N. Meisburger, M. Zhao, Y. Wu, S. Gobriel, C. Tai, and A. Shrivastava, "Accelerating slide deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more," 2021.

[58] M. E. Basiri, "Abcdm: An attention-based bidirectional cnn-rnn deep model for sentiment analysis," *Future Generation Computer Systems*, vol. 115, pp. 279–294, 2020.