Comparing Dataflow and OpenMP Programming for Speaker Recognition Applications

Hasna Bouraoui
Technische Universität Dresden, Germany
Dresden, Germany
hasna.bouraoui@tu-dresden.de

Jeronimo Castrillon
Technische Universität Dresden, Germany
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Chadlia Jerad
University of Manouba & University of Carthage
Tunis, Tunisia
chadlia.jerad@ensi-uma.tn

ABSTRACT

The still increasing number of transistors per chip offered by Moore's law, together with the Post-Dennard scaling era shifted the performance gain from frequency increase to multi-core processing. Consequently, the support of parallel execution of applications is becoming mandatory. Furthermore, the need for efficient parallel models and languages is more critical for the embedded domain, due to power consumption and memory constraints, among others. This work focuses on parallelizing an embedded speaker recognition application, which is a biometric technique for identification.

While a lot of work has been done for speech recognition, fewer efforts have focused on recognizing who the speaker is. In this paper, we analyze two implementations for speaker recognition applications (SRA), namely dataflow and shared memory programming models. More precisely, we use Process Networks (PNs) as a dataflow representation, which is an intuitive way to design streaming applications. We use the language "C for Process Networks" for the dataflow implementation and OpenMP for the shared memory one. For two different target platforms, we compared two implementations using OpenMP (exploring data-level parallelism only and with pipelining) against a dataflow-based compiled implementation that allows for functional optimization. Despite faster communication over shared memory, we show that the dataflow model is superior in terms of performance (up to twice as fast).

CCS CONCEPTS

• **Software and its engineering** → **Parallel programming languages**; *Data flow languages*; • **Theory of computation** → *Models of computation*; • **Computing methodologies** → Speech recog.

KEYWORDS

Dataflow models, Shared memory programming, Multicore programming, Speaker recognition

ACM Reference Format:
Hasna Bouraoui, Jeronimo Castrillon, and Chadlia Jerad. 2019. Comparing Dataflow and OpenMP Programming for Speaker Recognition Applications. In *Proceedings of PARMA-DITAM 2019 Workshop (PARMA-DITAM'19).* ACM, New York, NY, USA, 6 pages. https://doi.org/00.001/000_1

1 INTRODUCTION

According to Moore's law, the number of transistors in integrated circuits is doubling every 18 months. Mainly due to the break down of Dennard's scaling, the extra transistors are going into multi-core architectures, rather than into a single monolithic core. As a consequence, parallel programming stopped being a niche and became mainstream through many domains in computing, including embedded. Furthermore, as embedded computing has significant power budget constraints coupled with increasing needs for faster processing, parallel programming is mandatory.

We are interested in the parallelization of embedded signal processing applications for biometric authentication, that is, the process of automatically recognizing a person based on physical or personal behavior traits (e.g. iris, fingerprint, voice). Automatic speaker recognition, in particular, is generally related to the automatic identification of a person based on his vocal tract (i.e. voice). This could be used for many applications based on authentication such as in banking or building access control.

Many parallel programming models have been proposed in the last years, most of them in the area of desktop and high-performance computing. In the embedded domain, instead, there is a large body of research on *software synthesis* [10], with a strong focus on dataflow parallel programming models. These models are appropriate for describing intrinsically parallel processing over streams of data that flow from one node to the other. Therefore, these kinds of models are a good match for digital signal processing and multimedia applications. Furthermore, dataflow has been used for the synthesis and analysis of real-time embedded systems [5, 6] and automatic code generation from them has been investigated [8, 11].

Another prominent parallel programming model is OpenMP [23], a pragma-based approach for shared memory machines. OpenMP offers a good migration path for sequential code and includes a powerful runtime system to schedule threads in a transparent way to the programmer. In the embedded domain, OpenMP is poorly supported. For some platforms, the OpenMP stack alone consumes most of the resources, leaving only little for the application code and data. Parallel programs are thus still written by hand, using low-level application programming interfaces (APIs) like Pthreads.

In this paper, we investigate the expressiveness along with the performance, in terms of achieved speedup, of dataflow and shared memory implementations for speaker recognition applications. For the dataflow implementation, we use the MAPS dataflow framework (MPSOC Application Programming Studio), that uses an extension of the C language called CPN (C for Process Networks) [9]. For the shared memory implementation, we use OpenMP. As target systems, we use a 16-core desktop machine and an embedded

8-core Odroid board. We compare a first and intuitive implementation using OpenMP that explores data-level parallelism, against the dataflow version. Despite a faster communication over shared memory, the experimental results show that for the speaker recognition application, a dataflow programming approach achieves a better speedup. After modifying the OpenMP version to match the dataflow-like execution, the performance of the OpenMP version decreased even further. The experiments show that the OpenMP implementation based only on Data Level Parallelism (DLP), reach better speedup than the pipelined version.

The rest of this paper is organized as follows. In section 2 we discuss related work. Section 3 provides background on speaker recognition applications and Kahn Process Networks (KPNs). Section 4 presents the sequential and the OpenMP implementations. Section 5 describes the KPN model of the application, while section 6 compares the results and attempts to interpret the reason behind the performance gap between the different implementations. Finally, conclusions are drawn in Section 7.

2 RELATED WORK

Many works in the literature focus on accelerating speaker recognition process. This problem can be considered from different angles: modeling, software or hardware. From the modeling side, the speedup is achieved at the expense of accuracy [3, 4, 19, 25]. The lack of information about the execution platform and/or parallelization in these approaches let us conclude that the process was running sequentially on general purpose processors. From the hardware side, FPGA-based implementations, as well as custom hardware, have been used to accelerate speaker recognition applications. Readers may refer to [7] for a detailed survey.

In this paper we focus on a particular software implementation, exploiting the parallel processing capabilities of modern processors. This is enabled by parallel programming models, which, in an abstract way, allow shaping the parallel execution pattern of the application. To the best of our knowledge, only shared memory programming using OpenMP has been employed to accelerate, not speaker, but rather speech recognition applications [28]. Authors in [26] compare different programming models (OpenMP, GCD, and Pthreads) for speech and face recognition applications.

Others works have compared the performance of shared memory versus message passing implementations. Authors in [20] find no general conclusions w.r.t. performance. Factors like the load imbalance of the application as well as the communication overhead of the hardware have a great impact on the overall performance. Authors in [13] present the performance of 11 parallel benchmarks. They compare traditional shared memory against hybrid dataflow implementations, finding that dataflow offers higher flexibility than task-based models.

In our work, we explore and compare two different implementations of the target speaker recognition application, a shared-memory programming model (using OpenMP) and a dataflow-based model (using CPN).

3 BACKGROUND

In this section, we describe the general scheme of a speaker biometric recognition system. We then briefly introduce the model of computation used to specify the application as dataflow graph.

3.1 Speaker recognition applications

The generic process of speaker recognition is illustrated in Fig. 1. At the highest level, all speaker recognition systems contain two main modules: feature extraction and pattern matching. In the training phase, the speech utterances of known speakers are analyzed to extract their respective features. Speaker models are built based on these features and are stored in a database. The pattern matching phase consists in computing the similarity of an unknown speaker to the speakers' models in the database according to the speech signal analysis. The tested speaker model is compared against all the available models in the database. Once done, the system returns an identifier of the closest model to the input.

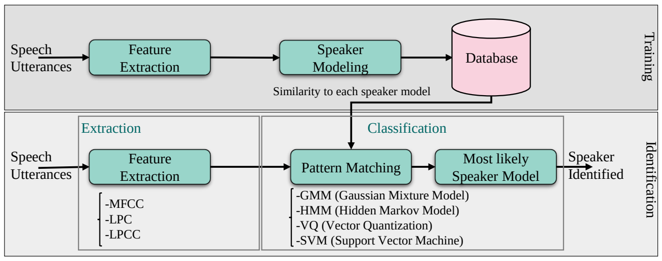

Figure 1: Speaker recognition process

Several approaches and algorithms have been used to perform recognition (as shown in Fig.1). Feature extraction can be performed by algorithms such as Reflection Coefficient (RCs), Linear Predictive Coding (LPC), Linear Prediction Cepstral Coefficient (LPCCs) or Mel-Frequency Cepstrum Coefficients (MFCCs). For the classification, the most used ones for speaker recognition include Hidden Markov Model (HMM), Dynamic Time Wrapping (DTW), Vector Quantization (VQ), Gaussian Mixture Model (GMM) and SVM (Support Vector Machine). Readers may refer to [7] for further details about the pros/cons and complexity of these algorithms.

3.2 Dataflow programming

A Model of Computation (MoC) depends on several regulations indicating how a concurrent execution of different components of the system and their mutual communication should be performed [24]. Such abstract models enable automated analysis that in turn enable automated optimization. For streaming applications such as SR, dataflow models, and/or their variants are well suited for describing them.

Kahn Process Networks (KPNs), or Process Networks (PNs) are an extension of dataflow models. A PN is a graph connecting concurrent processes (represented by nodes). The computation is divided among them and they communicate only through channels (represented by edges). When processes are arranged in a linear one directional chain, describing pipelined applications becomes implicit. Indeed, the use of buffers (channels) leads to pipelining in a natural and convenient way. The data flow between the processes, which makes them particularly suitable for describing streaming applications. Kahn showed in [15] that KPNs with unbounded FIFOs are deterministic, i.e., they always produce the same output, if provided with the same input.

For the implementation, we use the language "C for Process Networks" (CPN), an extension of the C language [8]. It adds new syntactic constructs to describe the dataflow graph. New keywords were added to describe Processes, Channels and Channel Accesses of the KPN dataflow model. Every process of the KPN graph executes code written in the standard C language. CPN is used as input to MAPS (MPSoC Application Programming Studio) compiler framework [9]. The framework generates optimized spatial and temporal process-to-core and logical to physical channel assignments for heterogeneous manycores. In our work, we use the commercially available SLX tool suite [2] for dataflow modeling and compilation.

4 SEQUENTIAL AND PARALLEL SHARED MEMORY IMPLEMENTATIONS

For our implementation of the speaker recognition, we use MFCC (Mel-Frequency Cepstrum Coefficient) [27] for feature extraction phase and VQ (Vector Quantization) [21] for the pattern matching. The overall flow of the speaker recognition is presented in Fig. 1.

For the feature extraction phase, the input speech utterance is divided into M overlapping frames of size N each (M being proportional to the utterance length). Then, and in order to avoid spectral effects, a Hamming window is applied to each of these frames that emphasizes higher frequencies. Afterward, these frames are transformed to the frequency domain by means of an FFT, just before a triangular filter bank is applied to estimate the human ear frequency. All M input frames are transformed into M acoustic vectors of a reduced size. Finally, applying DCT (Discrete Cosine Transform) transform on these vectors delivers the MFCC coefficients of each vector. Extracted vectors from speaker utterances are used to build the speaker model. This step is called the enrollment/training phase, after which all built models are stored in a database. Now, and to identify an unknown speaker utterance, the same steps, except for the enrollment phase, are needed. Then, the resulting model is compared to all the speaker models in the database. This comparison is performed by the similarity phase. The similarity is expressed in terms of a calculated distance, and the closest one to the unknown speaker is selected.

For a first application analysis, we profile the sequential implementation using Vtune[1]. The results showed that the pattern matching step is the most computationally intensive part, accounting for up to 97% of the CPU time. This computation has a regular structure with independent computations, i.e., it is highly data level parallel (DLP). Distances of the current speaker model to the set of saved models are computed respectively. Finally, the matching computation is also easily parallelized since it corresponds to a common reduction pattern.

Introducing DLP to sequential code can be done in an easy and intuitive way using OpenMP. The basic version of OpenMP implementation, which is a fully static one, is shown in Algorithm 1. The outer loop (line 6) iterates over the speakers to be recognized (`utterances`). The utterances are meant to be received on runtime, enabling thus stream processing. The computed model of the current utterance (`uttModel`) is fed to the inner loop (line 10), which iterates through all the models stored in the database, `dbSize` being the size of the database. The distance is calculated using the function `Similarity` (line 11), and the closest speaker model in terms of minimum distance is returned.

Here, the compiler OpenMP directive used at the outer loop is `pragma omp parallel` and at the inner loop is the `for` work-sharing construct with the `reduction` clause (line 9). The speaker variable in line 15 is shared and updated by concurrent threads. To ensure data integrity, we added the clause `omp critical`. Since `ComputeModel` (line 8) and `print` (line 19) should be executed once for each `utter`, we added two `omp single` clauses (lines 5 and 18).

In OpenMP, the `for` work-sharing construct supports different scheduling strategies, describing how and when iterations are assigned to threads. The aim is to adapt the behavior for reducing threads idle time. Since `Similarity` has a fixed size computation, the suitable schedule of the inner for loop is the default one, which is static with chunk size equal to the number of iterations divided by the number of threads.

We also considered another alternative, where we only annotate the inner loop with `pragma omp parallel for reduction`. This version, however, performs sometimes worse due to the additional overhead of repeatedly opening and closing the parallel region.

ALGORITHM 1
Speakers_Recognition_version_1

```
1:  procedure PARALLELSRA($utterances, dbSize$)
2:      $database \leftarrow ReadDatabase(dbSize)$
3:      $distMin \leftarrow 9999$          ▷ Initialize the minimal distance to an upper value
4:      $speaker \leftarrow null$          ▷ Recognized speaker in the database
5:      #pragma omp parallel
6:      for $utter$ in $utterances$ do
7:          #pragma omp single private(utter)
8:          $uttModel \leftarrow ComputeModel(utter)$
9:          #pragma omp for reduction(min:distMin)
10:         for $mInDB = 1$ to $dbSize$ do
11:             $dist \leftarrow Similarity(uttModel, database[mInDB])$
12:             #pragma omp critical
13:             if $dist < distMin$ then
14:                 $distMin \leftarrow dist$
15:                 $speaker \leftarrow mInDB$
16:             end if
17:         end for
18:         #pragma omp single
19:         $print(speaker)$               ▷ Print the identified speaker
20:     end for
21: end procedure
```

5 DATAFLOW IMPLEMENTATION

For the implementation of the SRA dataflow we use the SLX tool suite. After adding CPN annotations to the sequential C code, we generate the dataflow shown in Fig. 2. The communication channels sizes are statically set by the tool, before generating the runtime.

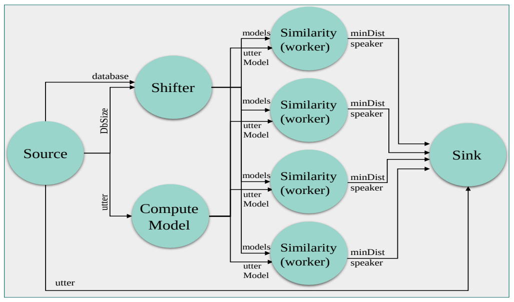

Figure 2: Process network of the speaker recognition application.

[1]https://software.intel.com/en-us/get-started-with-vtune

Speaker identification, presented by the graph in this figure, starts at the *Source* node. This node loads the speaker models stored in the database and sends them (*database*) to the *Shifter* node. The number of the tokens is the number of models in the database, while each one is a 16×19 matrix of doubles model. The *Shifter* distributes the tokens among *Similarity*(*Worker*) nodes while ensuring a balanced distribution (in terms of number). The number of workers has been changed throughout the implementation, to see the behavior of different parallel granularities. *Source* node loads, in addition, the utterances of the speakers to be recognized and sends them (*utter*) to *ComputeModel* node. *ComputeModel* creates the respective models of the speaker utterances to be recognized. First, features are extracted and then form the utterance model (*uttModel*). As soon as the first model (*uttModel*) is created by the *ComputeModel* node, it is sent to all *Similarity*(*Worker*) nodes to compare it against the models in the database (each node will compare it to the speaker models it received from the *Shifter*). The total number of sent *utterModel* tokens is equal to the number of speakers to recognize. This behavior describes the implicit pipelining in the dataflow. While *Similarity*(*Worker*) node is working on the current speaker, the previous nodes are computing the model of the next utterance to recognize. For each *Similarity*(*Worker*) node, the closest model will be returned to *Sink* which will print the speaker ID with the smallest distance. Given that the worker nodes run in parallel, the CPN model features the same data level parallelism that we expressed in the OpenMP model (cf. Algorithm 1).

6 EVALUATION

For the experiments part, we deploy the speaker recognition application on two different platforms. In this section, we compare the dataflow implementation discussed in Section 5 with the OpenMP variant presented in section 4. And we analyze another openMP implementation, that mimics the behavior of the CPN implementation.

6.1 Experimental setup

To compare the results, we use two different platforms. The desktop platform is an x86-64 ISA general purpose platform (denoted hereafter GPP). It has an Intel Xeon Processor X5550 with 16 cores and 32GB of memory. The operating frequency of the processor is 1.5 GHz (and up to 2.6 GHz). For the embedded platform, we use Odroid XU4 [1] board. This is an octa-core board with 4 Cortex A15 cores and 4 Cortex A7 cores. It has a 2 Gbyte LPDDR3 DRAM memory. The frequencies are 1.4 GHz and 2.0 GHz. Each core has an L1 cache memory level of 32KB. Besides, the 4 Cortex A15 and Cortex A7 share an L2 cache memory level of 2Mb and 512KB respectively.

As a benchmark, recognition experiments are performed on the TIMIT corpus [14]. TIMIT contains 630 speakers where 70% are men and 30% are women.

For the experimental results, the input to the SRA is audio utterances of these speakers. Models extracted from these speech utterances are represented by a double-precision 16x19 matrix. In addition to the database, the SR application takes as input speaker utterances with a number of frames laying between 408 and 796 each. Every frame is of size 256 (25 ms) sampled at 16 kHz overlapped of 100 samples (16 ms).

6.2 CPN-OpenMP comparison

For comparing CPN and OpenMP, we want to explore different possibilities. We do this by modifying the following parameters: (i) the platforms: GPP and Odroid, (ii) the size of the database: 2000, 8000 and 16000 speakers, (iii) the number of the speakers to recognize: 1, 5 and 10, (iv) and the number of workers computing the similarity (in the case of the CPN implementation), the number of threads (in the case of the OpenMP implementation) and combined with the number of cores to run on.

Fig.3 reports the speedup obtained with the OpenMP and the CPN versions w.r.t. the sequential implementation on both platforms (GPP and Odroid). On GPP, we achieve a higher speedup, for a bigger number of speakers to recognize, also while maintaining the same database size, the same number of workers/threads and cores ((a) refers to 2 workers/threads on 4 cores, (b) refers to 4 workers/threads on 8 cores and (c) refers to 8 workers/threads on 8 cores). The experiments ran on Odroid platform showed similar results. For space reasons, we only show the achieved speedup for 5 speakers in the same Fig.3.

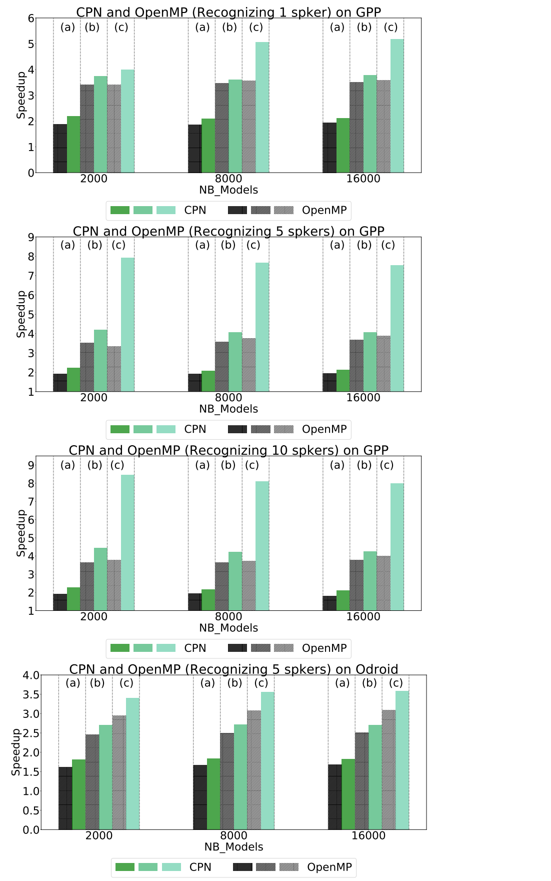

Figure 3: Comparing speedup of CPN and DLP-only OpenMP implementations on GPP recognizing 1 , 5 and 10 speakers and on Odroid recognizing 5 speakers.(a) 2 workers/threads on 4 cores (b) 4 workers/threads on 8 cores (c) 8 workers/threads on 8 cores

All the experimental results revealed that the dataflow implementation performs better (in terms of achieved speedup) than the shared memory implementation using OpenMP. This is unexpected, considering the overhead involved in managing the FIFO communication in the generated code from CPN. In order to understand the reason behind these results, we investigated the execution schema of both implementations. For clarity reasons, we show the Gantt charts of the executions using 4 workers on 6 cores for CPN and 4 threads for OpenMP. Both charts consider more than one speaker to recognize.

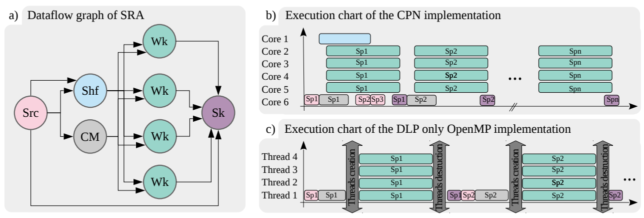

Figure 4: Gantt chart graphs. b) Execution graph of the CPN implementation. c) Execution graph of the OpenMP implementation (only DLP support).

The dataflow model behavior, shown in Fig.4 (based on the mapping of the nodes to the cores that is generated by the SLX tool suite), reflects the parallel behavior outlined by the dataflow. However, the shared memory model behavior outlines the DLP of the inner loop, leading to a considerable overhead. This overhead is due to synchronization and runtime jobs/threads management. Indeed authors in [12] showed that OpenMP overheads increase in importance with the number of cores, while authors in [22] demonstrated that OpenMP 3.0 implementations exhibit poor behavior. Additionally, as shown in Fig.4, the CPN version exposes more parallelism than the OpenMP version does, due to the pipelining enabled via the buffering in the communication channels. The next section analyses a more involved OpenMP version of the code that mimics the behavior of the CPN implementation.

6.3 Analysis of a dataflow-like OpenMP implementation

To mimic the dataflow execution with OpenMP, we use a topmost parallel region with the entire for loop being executed as single and use tasks inside. Coming up with such solution required to go through a complex structure and nesting of the tasks. Main additions to the parallel version that uses OpenMP with DLP support only are colored in Algorithm 2. In this pseudocode, statements in purple are OpenMP directives, while statements in blue highlight the extra needed variables and processing. In this alternative, tasks are created and scheduled at runtime, leading thus to dynamic management compared to the static one in the first implementation.

As depicted in Algorithm 2, one main task (`single` thread in line 5) iterates over the utterances to recognize. For each utterance (i.e. each iteration), it creates a separate *big* task (line 8 to line 33). We notice here that whenever a big task is computing the similarity of the current model in the current iteration, the main task starts creating the next utterance model. This enables parallelism between these two computations.

The *big* task structure is also complex. It starts computing the dala level parallelism by applying the `Similarity()` function on a chunk of the models existing in the database. Each chunk computation is ensured by a newly created subtask (line 15). In this case, within each of these *big* tasks, `nbrWorkers` (i.e. line 3) subtasks are created to execute this similarity function in parallel (i.e. DLP) and synchronize by the end.

The chunk size is defined at runtime by two variables: `istart` (lines 12 and 25) and `iend` (lines 13 and 26). Before applying the reduction (line 30), the created subtasks need to synchronize. This synchronization is ensured by the `taskwait` construct (line 28) that imposes a barrier. In fact, the obtained runtime corresponds to as many created *big* tasks as the given number of utterances. Despite the fact that they are created in deferred instants, they still can be scheduled in parallel. Finally, an additional subtask is created by the main task (line 29 to 32) to perform the reduction and results displaying.

Algorithm 2
Speakers_Recognition_version_2

```
 1: procedure PARALLELSRA_CPNLIKE($utterances, dbSize$)
 2:     $database \leftarrow ReadDatabase(dbSize)$
 3:     $nbrWorkers \leftarrow 4$
 4:     #pragma omp parallel {
 5:         #pragma omp single {
 6:             for $utter$ in $utterances$ do
 7:                 $uttModel \leftarrow ComputeModel(utter)$
 8:                 #pragma omp task firstprivate($model$) {
 9:                                 ▷ distMin and speaker are tables of nbrWorkers local minimums
10:                     $[nbrWorkers]distMin \leftarrow [9999, 9999, \ldots]$
11:                     $[nbrWorkers]speaker \leftarrow [null, null, \ldots]$
12:                     $istart \leftarrow 0$   ▷ Explicitly compute the the model indexes of each worker
13:                     $iend \leftarrow dbSize/nbrWorkers$
14:                     for $wkInd = 1$ to $nbrWorkers$ do
15:                         #pragma omp task firstprivate (dist, istart, iend) {
16:                             for $mInDB = istart$ to $iend$ do
17:                                 $dist \leftarrow Similarity(uttModel, database[mInDB])$
18:                                 #pragma omp critical
19:                                 if $dist < distMin[wkInd]$ then
20:                                     $distMin[wkInd] \leftarrow dist$
21:                                     $speaker[wkInd] \leftarrow mInDB$
22:                                 end if
23:                             end for
24:                         }
25:                         $istart \leftarrow iend$
26:                         $iend \leftarrow iend + dbSize/nbrWorkers$
27:                     end for
28:                     #pragma omp taskwait                ▷ synchronize distance computation
29:                     #pragma omp task {
30:                         $reduce(distMin, speaker)$
31:                         $print(speaker)$                   ▷ Print the identified speaker
32:                     }
33:                 }
34:             end for
35:         }
36:     }
37: end procedure
```

Getting a CPN-like behavior using OpenMP was achieved but at the expense of a completely non-intuitive new OpenMP implementation. For the experimental results, we change the same parameters for both implementations (i.e number of threads, number of workers, etc.).

We compare in Fig.5 the old (and static) OpenMP version speedup with the CPN-like (and dynamic) one while varying the hardware platform. The experimental results show that the old OpenMP implementation (with DLP support only) achieves better speedup than the new one on both platforms. Therefore, CPN implementation still achieves better speedup. Besides being complex and not intuitive, the overhead of the `task` construct is remarkable. The CPN implementation is superior, justifying the extra effort in using the right MoC for the task at hand.

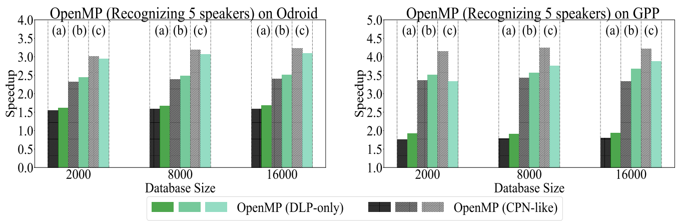

Figure 5: Comparing speedup of CPN-like and DLP-only OpenMP implementations recognizing 5 speakers on GPP and Odroid platform. (a) 2 threads on 4 cores (b) 4 threads on 8 cores (c) 8 threads on 8 cores.

7 DISCUSSION AND CONCLUSION

Model-based programming helps to address interactions schemes over the system components in an abstract manner. In this paper, we analyze dataflow and shared memory programming models for speaker recognition applications. We first compare the dataflow implementation using CPN with an intuitive shared memory implementation, yet static, using OpenMP. For two different target platforms, a notable difference in achieved speedup in the experimental results shows that despite a faster communication over shared memory, the OpenMP runtime adds significant overhead, that is partially due to synchronization and runtime management. Analyzing the Gantt Charts of both implementations, we went for a fairer comparison by mimicking dataflow behavior in the shared memory implementation. This version performs worse in all cases, due to the overhead of dynamic task management. The CPN implementation is superior, despite the slower communication using channels.

In future work, we plan to further exploit the possibilities that the dataflow model-based approach offers for speaker recognition. We could exploit the symmetry-enabled predictable execution of static mappings introduced in [16, 17]. We could also improve the run-time adaptability on a heterogeneous platform by providing implicit parallelism of the KPN, as shown in [18]. Finally, we could vary the number of threads during runtime with the different levels of data parallelism.

ACKNOWLEDGEMENTS

This work was partially supported by the German Research Council (DFG) through the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed).

REFERENCES

[1] [n. d.]. Ordoid XU4. http://magazine.odroid.com/odroid-xu4/ Accessed: 2017-02-03.
[2] [n. d.]. Silexica. https://www.silexica.com/ Accessed: 2018-05-9.
[3] Vijendra Raj Apsingekar and Phillip L De Leon. 2008. Efficient speaker identification using speaker model clustering. In *Signal Processing Conference,16th European*. IEEE.
[4] Roland Auckenthaler and John S Mason. 2001. Gaussian selection applied to text-independent speaker verification. In *2001: A Speaker Odyssey-The Speaker Recognition Workshop*.
[5] Marco Bekooij, Rob Hoes, Orlando Moreira, Peter Poplavko, Milan Pastrnak, Bart Mesman, Jan David Mol, Sander Stuijk, Valentin Gheorghita, and Jef Van Meerbergen. 2005. Dataflow analysis for real-time embedded multiprocessor system design. In *Dynamic and robust streaming in and between connected consumer-electronic devices*. Springer, 81–108.
[6] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. 1999. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology* 21, 2 (1999), 151–166.
[7] Hasna Bouraoui, Chadlia Jerad, Anupam Chattopadhyay, and Nejib Ben Hadj Alouane. 2016. Hardware Architectures for Embedded Speaker Recognition Applications-a Survey. *to appear in ACM Transactions on Embedded Computing* (2016).
[8] Jeronimo Castrillon and Rainer Leupers. 2014. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap.* Springer. 258 pages.
[9] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. 2013. MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics* 9, 1 (Feb. 2013), 527–545.
[10] Jeronimo Castrillon, Weihua Sheng, and Rainer Leupers. 2011. Trends in Embedded Software Synthesis. In *Proceedings of the International Conference Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS), 2011*. IEEE, 347–354.
[11] Johan Eker and Jörn W Janneck. 2003. *CAL language report: Specification of the CAL actor language.* Electronics Research Laboratory, College of Engineering, University of California.
[12] Karl Fürlinger and Michael Gerndt. 2006. Analyzing overheads and scalability characteristics of OpenMP applications. In *International Conference on High Performance Computing for Computational Science*. Springer, 39–51.
[13] Vladimir Gajinov, Srdjan Stipić, Igor Erić, Osman S Unsal, Eduard Ayguadé, and Adrian Cristal. 2015. DaSH: A benchmark suite for hybrid dataflow and shared memory programming models. *Parallel Comput.* 45 (2015), 18–48.
[14] John S Garofolo et al. 1988. Getting started with the DARPA TIMIT CD-ROM: An acoustic phonetic continuous speech database. *National Institute of Standards and Technology (NIST), Gaithersburgh, MD* 107 (1988), 16.
[15] KAHN Gilles. 1974. The semantics of a simple language for parallel programming. *In Information Processing* 74 (1974), 471–475.
[16] Andrés Goens, Robert Khasanov, Jeronimo Castrillon, Marcus Hähnel, Till Smejkal, and Hermann Härtig. 2017. Tetris: A multi-application run-time system for predictable execution of static mappings. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*. ACM, 11–20.
[17] Andrés Goens, Sergio Siccha, and Jeronimo Castrillon. 2017. Symmetry in Software Synthesis. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2, Article 20 (July 2017), 26 pages. https://doi.org/10.1145/3095747
[18] Robert Khasanov, Andrés Goens, and Jeronimo Castrillon. 2018. Implicit Data-Parallelism in Kahn Process Networks: Bridging the MacQueen Gap. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. ACM, 20–25.
[19] Tomi Kinnunen, Evgeny Karpov, and Pasi Franti. 2006. Real-time speaker identification and verification. *IEEE Transactions on Audio, Speech, and Language Processing* 14, 1 (2006), 277–288.
[20] Thomas J LeBlanc and Evangelos P Markatos. 1992. Shared memory vs. message passing in shared-memory multiprocessors. In *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on*. IEEE, 254–263.
[21] John Makhoul, Salim Roucos, and Herbert Gish. 1985. Vector quantization in speech coding. *Proc. IEEE* 73, 11 (1985), 1551–1588.
[22] Stephen L Olivier and Jan F Prins. 2010. Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming* 38, 5-6 (2010), 341–360.
[23] ARB OpenMP. 2013. OpenMP 4.0 specification, June 2013. http://www.openmp.org/
[24] Claudius Ptolemaeus. 2014. *System Design, Modeling, and Simulation using Ptolemy II, 2014.* Ptolemy.org. http://ptolemy.org/systems
[25] Bing Sun, Wenju Liu, and Qiuhai Zhong. 2003. Hierarchical speaker identification using speaker clustering. In *Natural Language Processing and Knowledge Engineering, 2003. Proceedings. 2003 International Conference on*. IEEE, 299–304.
[26] Deepak Shekhar TC, Kiran Varaganti, Rahul Suresh, Rahul Garg, and Ramalingam Ramamoorthy. 2011. Comparison of parallel programming models for multicore architectures. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 1675–1682.
[27] Roberto Togneri and Daniel Pullella. 2011. An overview of speaker identification: Accuracy and robustness issues. *IEEE Circuits and Systems Magazine* 11, 2 (2011), 23–61.
[28] Kisun You, Youngjoon Lee, and Wonyong Sung. 2009. OpenMP-based parallel implementation of a continuous speech recognizer on a multi-core system. In *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 621–624.