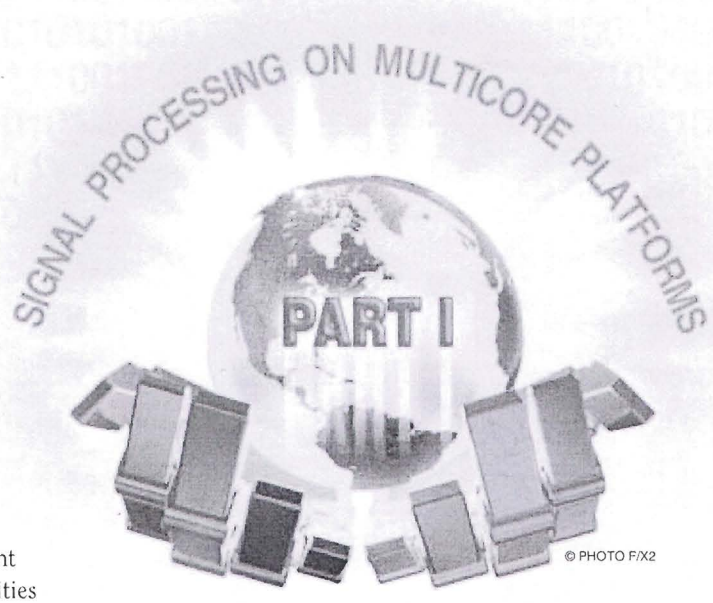
Kisun You, Jike Chong, Youngmin Yi, Ekaterina Gonina,
Christopher J. Hughes, Yen-Kuang Chen, Wonyong Sung, and Kurt Keutzer

Parallel Scalability in Speech Recognition

Inference engines in large vocabulary continuous speech recognition



Parallel scalability allows an application to efficiently utilize an increasing number of processing elements. In this article, we explore a design space for parallel scalability for an inference engine in large vocabulary continuous speech recognition (LVCSR). Our implementation of the inference engine involves a parallel graph traversal through an irregular graph-based knowledge network with millions of states and arcs. The challenge is not only to define a software architecture that exposes sufficient fine-grained application concurrency but also to efficiently synchronize between an increasing number of concurrent tasks and to effectively utilize parallelism opportunities in today's highly parallel processors.

We propose four application-level implementation alternatives called algorithm styles and construct highly optimized implementations on two parallel platforms: an Intel Core i7 multicore processor and a NVIDIA GTX280 manycore processor. The highest performing algorithm style varies with the implementation platform. On a 44-min speech data set, we demonstrate substantial speedups of 3.4× on Core i7 and 10.5× on GTX280 compared to a highly optimized sequential implementation on Core i7 without sacrificing accuracy. The parallel implementations contain less than 2.5% sequential overhead, promising scalability and significant potential for further speedup on future platforms.

Digital Object Identifier 10.1109/MSP.2009.934124

INTRODUCTION

We have entered a new era where sequential programs can no longer fully exploit a doubling in scale of integration according to Moore's law [1]. Parallel scalability, the ability for an application to efficiently utilize an increasing number of processing elements, is now required for software to obtain sustained performance improvements on successive generations of processors.

Many modern signal processing applications are evolving to incorporate recognition backends that have significant scalability challenges. In this article, we examine the scalability challenges in implementing a hidden Markov model (HMM)-based

1053-5888/09/$26.00©2009IEEE

inference algorithm in an LVCSR application.

An LVCSR application analyzes a human utterance from a sequence of input audio waveforms to interpret and distinguish the words and sentences intended by the speaker. Its top level architecture is shown in Figure 1. The recognition process uses a recognition network, which is a language database that is compiled offline from a variety of knowledge sources trained using powerful statistical learning techniques. The speech feature extractor collects feature vectors from input audio waveforms using standard scalable signal processing techniques [2], [3] and is not discussed in this article. The inference engine traverses a graph-based recognition network based on the Viterbi search algorithm [4] and infers the most likely word sequence based on the extracted speech features and the recognition network. In a typical recognition process, there are significant parallelism opportunities in concurrently evaluating thousands of alternative interpretations of a speech utterance to find the most likely interpretation. We explore these opportunities in detail in this article.

> PARALLEL SCALABILITY IS NOW REQUIRED FOR SOFTWARE TO OBTAIN SUSTAINED PERFORMANCE IMPROVEMENTS.

Parallel graph traversal on large unstructured graphs is a well-known challenge for scalable parallel computation [5], especially in the context of an LVCSR inference engine [6]. The traversal is conducted over an irregular graph-based knowledge network and is controlled by a sequence of audio features known only at run time. Furthermore, the data working set changes dynamically during the traversal process, and the algorithm requires frequent communication between concurrent tasks. These problem characteristics lead to unpredictable memory accesses and poor data locality and cause significant challenges in load balancing and efficient synchronization between processor cores.

In this article, we demonstrate the implications of these challenges on two highly parallel architectures: an Intel Core i7 multicore processor and an NVIDIA GTX280 manycore processor. We consider multicore processors as processors that devote significant transistor resources to complex features for accelerating single thread performance, whereas manycore processors use their transistor resources to maximize total instruction throughput at the expense of single thread performance. We show that the best algorithm on one architecture may perform poorly on another due to varying efficiencies of key parallel operations, and that the efficiency of the key parallel operations is more indicative of the performance of the application implementation.

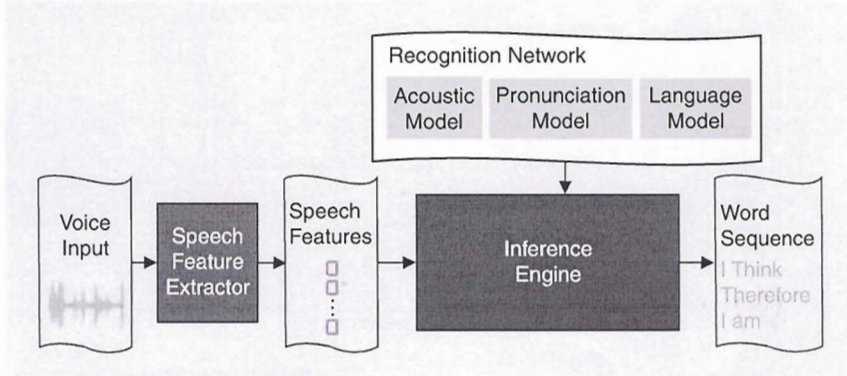
We discuss two important issues in multicore and manycore programming: exploiting single-instruction, multiple-data (SIMD) parallelism and implementing efficient synchronization between cores. SIMD execution involves simultaneously computing multiple data elements in parallel lanes of functional units. SIMD efficiency is a measure of how well an algorithm can make use of functional units with a certain number of lanes (i.e., a given SIMD width). For algorithms with a lot of data parallelism (including those examined here) high SIMD efficiency at a given SIMD width indicates that the algorithm is likely to benefit greatly from an even wider SIMD. At the core level, synchronization between cores incurs long latencies and limits throughput. Efficient synchronization between cores reduces the management overhead of a parallel algorithm and allows the same problem to gain additional speedups as we scale to more cores. We set up four algorithm styles to compare two graph traversal techniques for efficient SIMD utilization and two coordination techniques for core-level synchronization. We show that differences in features of a platform's microarchitecture can lead to very different optimal configurations of the inference algorithm.

RELATED WORK

There have been many attempts to parallelize speech recognition on emerging platforms, leveraging both fine-grained and coarse-grained concurrency in the application.

Ravishankar in [7] mapped fine-grained concurrency onto the PLUS multiprocessor with distributed memory. The implementation statically mapped a carefully partitioned recognition network onto the multiprocessors to minimize load imbalance. While achieving 3.8× speedup over the sequential implementation on five processors, the static partitioning would not scale well to 30+ cores because of load imbalance at run time. Agaram et al. showed an implementation of LVCSR on a multiprocessor simulator [8]. However, the simulator did not model the synchronization overhead between cores, which is crucial for scalability analysis.

Ishikawa et al. [9] explored coarse-grained concurrency in LVCSR and implemented a pipeline of tasks on a cellphone-oriented multicore architecture. They achieved 2.6× speedup over a sequential baseline version by distributing tasks among three ARM cores. However, it is difficult for this implementation to scale beyond three cores due to the a small amount of function-level concurrency in the algorithm.



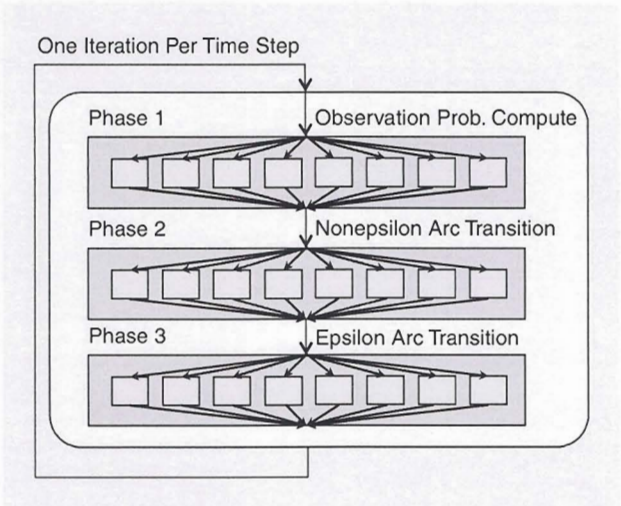
[FIG1] Architecture of an LVCSR application.

You et al. [10] have recently proposed a parallel LVCSR implementation on a commodity multicore system using OpenMP. The Viterbi search was parallelized by statically partitioning a tree-lexical search network across cores. However, only 2× speedup was achieved on shared-memory Intel Core2 quadcore processors due to limited memory bandwidth. A tree-lexical search-based inference engine is tightly coupled with recognition network features: many improvements in the network compilation techniques require corresponding changes in the inference engine. We address this with a different weighted finite state transducer (WFST)-based recognition network in this article (see the section "Characteristics of LVCSR").

The parallel LVCSR system proposed by Phillips et al. also uses WFST and data parallelism when traversing the recognition network [11]. They achieved 4.6–6.2× speedup on 16 processors, however, this implementation was limited by sequential components in the recognizer and load imbalance among processors. Their private buffer-based synchronization imposes significant data structure overhead and is not scalable with the increasing number of cores.

Prior works such as [12] and [13] by Dixon et al. and Cardinal et al. leveraged manycore processors and focused on speeding up the compute-intensive phase (i.e., observation probability computation) of LVCSR on manycore accelerators. Both [12] and [13] demonstrated approximately 5× speedups in the compute-intensive phase and mapped the communication intensive phases (i.e., Viterbi search) onto the host processor. This software architecture incurs significant penalty for copying intermediate results between the host and the accelerator subsystem and does not expose the maximum potential of the performance capabilities of the platform.

Chong et al. in [14] implemented a data parallel LVCSR on the NVIDIA 8800 GTX for a linear lexicon-based recognition network. Leveraging the regular structure of the network, they achieved a 9× speedup compared to a SIMD optimized sequential implementation on Core2 CPU. This linear lexical-based implementation is highly optimized for a simple language model and cannot easily incorporate advanced language model features without incurring significant performance penalties. The WFST approach in this article addresses this issue.



[FIG2] **Software architecture of the LVCSR inference engine. Multiple steps in a phase, each has 1,000–10,000 s concurrent tasks.**

In this work, we optimize the software architecture for highly parallel multicore and manycore platforms, explore multiple scalable synchronization methods, and traverse the more challenging WFST-based recognition network. For the manycore implementation, we implement both computation intensive and communication intensive phases on the manycore platform thereby eliminating expensive data-copying penalty of intermediate results between the host and the manycore accelerator.

CHARACTERISTICS OF LVCSR

Speech recognition is the process of interpreting words from speech waveforms. The simplest problem is an isolated word recognition task, such as discriminating between a "yes" or a "no" in an interactive voice response system. Such tasks have small vocabularies that can be searched exhaustively and can generally be solved with modest computation effort.

In contrast, LVCSR is a more difficult problem. For example, the objective might be to provide a transcription for a video sequence. The LVCSR system must be able to recognize words from a very large vocabulary arranged in exponentially many permutations and with unknown boundary segmentation between words. Mathematically, it finds the most probable word sequence $\hat{W}$ for a sequence of observed audio features O given the set of possible word sequences W as follows:

$$\hat{W} = \arg\max_{W} \{P(O|W)P(W)\}. \quad (1)$$

The product of acoustic and prior likelihood for the word sequence W, i.e., $P(O|W)P(W)$ is computed using a dynamic programming recurrence in the Viterbi search algorithm [4].

The likelihood of the traversal process being in state j with word sequence w_{ij} at time t can be derived from the likelihood in preceding states as follows:

$$\psi_t(s_j; w_{ij}) = \max_i \{\psi_{t-1}(s_i; w_{(t-1)i}) \cdot a_{ij} \cdot b(O_t; m_k)\}, \quad (2)$$

where a_{ij} is a transition probability from state i (s_i) to state j (s_j), and $b(O_t; m_k)$ is the observation probability of context-dependent state k (m_k) on transition from s_i to s_j. The algorithm iterates over a sequence of time steps. The likelihood of a word sequence in each time step depends on the likelihood computed in the previous time step. We refer to this as the iterations of the inference engine (Figure 2). In each iteration we maintain thousands of active states, that represent the most likely alternative interpretations of the input speech waveforms and select the most likely interpretation at the end of a speech utterance.

The WFST approach has been recently adopted as the primary recognition network representation used in speech

recognition algorithms [15]. A WFST is a Mealy finite state machine (FSM) represented by a list of arcs with five properties: source state, destination state, input symbol, output symbol, and weight. The recognition network usually consists of four hierarchical knowledge sources: HMM acoustic model H, context model C, pronunciation lexicon of words L, and language model G that can be composed into one $H \circ C \circ L \circ G$ WFST, also known as the H-level network. The combined WFST can be optimized using standard FSM minimization techniques described in [15] and used as a flattened FSM representation for the recognition network.

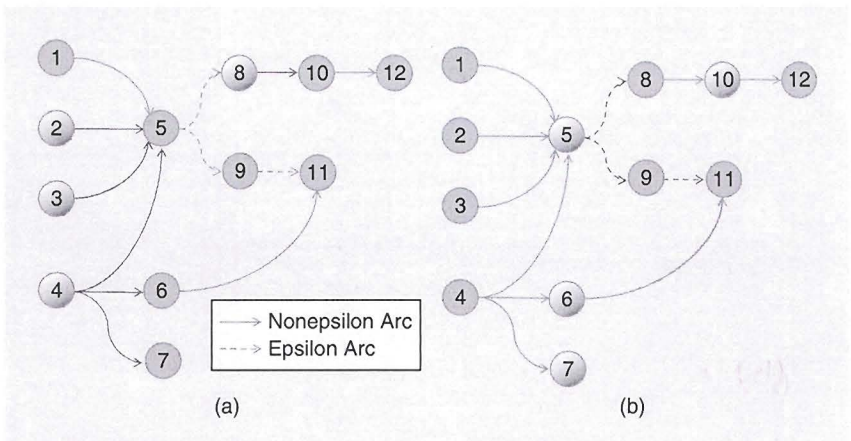
WFST has several advantages. First, it greatly simplifies the recognition procedure by flattening the hierarchical knowledge sources offline into a single level FSM to be traversed at run time. Second, WFST-based search is known to be more efficient than other search methods in terms of necessary computation. Detailed comparison in [16] shows that the WFST-based search is faster than the tree-lexical search, since it explores fewer search states for a given word error rate (WER). Finally, the WFST-based inference engine is application agnostic: it can be employed in other domains such as text and image processing [15].

Figure 3 shows the graph traversal process on a section of a WFST-based recognition network. There are two types of arcs: nonepsilon arcs and epsilon arcs. The nonepsilon arcs consume one input symbol to perform a state transition while epsilon arcs are traversed without consuming any input symbols. Since we utilized a H-level WFST recognition network, the input labels of the graph represent the context-dependent HMM states. Figure 2 illustrates the architecture of the recognition algorithm. For each input frame, the recognizer iteratively traverses the recognition network in the following three phases:

- *Phase 1: Observation probability computation.* The observation probability measures the likelihood of an input feature matching an acoustic input symbol (the Gaussian mixture model of a context-dependent state) by computing a distance function. Only the input symbols on the active arcs (i.e., the outgoing arcs of active states) need to be computed. This phase references the acoustic models shown in Figure 1.
- *Phase 2: Nonepsilon arc transitions.* The nonepsilon arc transitions shown in Figure 3(a) compute a joint probability of three components shown in (2). These components are 1) the observation probability of the current input $b(O_t; m_k)$ computed in Phase 1, 2) the transition probability or the weight of the arc being traversed a_{ij} referenced from the WFST recognition network, and 3) the likelihood of prior sequences, or the source state cost $\psi_{t-1}(s_i; w_{(t-1)i})$ computed in the previous iteration at time $t-1$. The result is the product of the three components. Due to the Viterbi approximation, the cost of a destination state is updated with the cost of the most likely incoming nonepsilon arcs for that state.
- *Phase 3: Epsilon arc transitions.* The epsilon arcs do not have input symbols, so the probabilities are computed as the product of two components: 1) the transition probability, and 2) the likelihood of prior sequences. The network might contain a chain of consecutive epsilon arcs, as shown in Figure 3(b). By definition of epsilon arcs we must traverse all outgoing epsilon arcs from each destination state until we reach a state with no outgoing epsilon arcs. This phase references the WFST recognition network.

Among the three phases, Phase 1 is the compute-intensive phase, where more than 90% of the computation is spent in evaluating the Gaussian mixture model in the observation probability computation. Phases 2 and 3 are the communication-intensive phases. In these phases, the computation involves aggregating multiple components from different sources with different parallelization granularities and with intermediate results extensively communicated between parallel processing units.

As shown in Figure 2, despite the fact that the recognition procedure for each frame is sequential in nature, each phase of the recognition has significant opportunities for fine-grained parallelism. Thousands of acoustic input symbols are utilized to compute the observation probability in Phase 1, and tens of thousands of arc transitions are traversed through the WFST network in Phases 2 and 3. This presents an opportunity for fine-grained concurrency in the LVCSR inference engine. We need to scalably exploit the parallelism of each step to gain performance on multicore and manycore platforms.



[FIG3] **Graph traversal in a WFST-based recognition network.**

ALGORITHM STYLES OF THE INFERENCE ENGINE

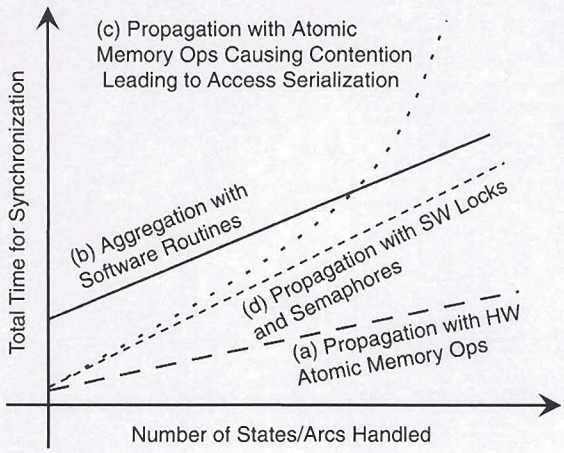
Given the challenging and unpredictable nature of the underlying graph-traversal algorithm in LVCSR, implementing it on parallel platforms presents two architectural challenges: efficient core level synchronization and efficient SIMD utilization. These challenges are key factors in making the algorithms scalable to increasing number of cores and SIMD lanes in multicore and manycore platforms. To find a solution to these challenges, we explore two aspects of the algorithmic level design space: the graph traversal technique and the recognition network transition evaluation granularity. Our design space is shown in Figure 4.

> HIGH SIMD EFFICIENCY ENABLES THE ALGORITHM TO BENEFIT FROM AN EVEN WIDER SIMD UNIT IN FUTURE PROCESSORS.

Algorithm Styles in the Design Space

Transition Evaluation Granularity / Addressing SIMD Utilization	Arc Based One Arc at a Time	Arc-Based Aggregation Approach	Arc-Based Propagation Approach
	State Based All Outgoing/ Incoming Arcs at a State	State-Based Aggregation Approach	State-Based Propagation Approach
		Aggregation Traversal Organized at Destination State	Propagation Traversal Organized at Source State
		Graph Traversal Techniques Addressing Core-Level Synchronization	

[FIG4] **The algorithmic level design space for graph traversal scalability analysis for the inference engine.**



[FIG5] **Scalability of the traversal process in terms of total synchronization time.**

GRAPH TRAVERSAL TECHNIQUES

One can organize the graph traversal in two ways: by propagation or aggregation. During the graph traversal process, each arc has a source state and a destination state. Traversal by propagation organizes the traversal process at the source state. It evaluates the outgoing arcs of the active states and propagates the result to the destination states. As multiple arcs may be writing their result to the same destination state, this technique requires write conflict resolution support in the underlying platform. Traversal by aggregation organizes the traversal process around the destination state. The destination states update their own information by performing a reduction on the evaluation results of their incoming arcs. This process explicitly manages the potential write conflicts by using additional algorithmic steps such that no write conflict resolution support is required in the underlying platform.

The choice of the traversal technique has direct implications on the cost of core level synchronization. Efficient synchronization between cores reduces the management overhead of a parallel algorithm and allows the same problem to gain additional speedups as we scale to more cores. Furthermore, there are additional implications on design productivity and code portability for parallel implementations.

SYNCHRONIZATION EFFICIENCY

Minimizing the total cost of synchronization is key to making the traversal process scalable. Figure 5 outlines the tradeoffs in the total cost of synchronization between the aggregation technique and the propagation technique. The qualitative graph shows increasing synchronization cost with increasing number of concurrent states or arcs evaluated.

The fixed cost for the aggregation technique (Y-intercept of line (b) in Figure 5) is higher than that of the propagation technique, as it requires a larger data structure and a more complex set of software routines to manage potential write conflicts. The relative gradient of the aggregation and propagation techniques depends on the efficiency of the platform in resolving potential write conflicts. If efficient hardware-supported atomic operations are used, the variable cost for each additional access would be small, and the propagation technique should scale as line (a) in Figure 5. If there is no hardware support for atomic operations and sophisticated semaphores and more expensive software-based locking routines are used, the propagation technique would scale as line (d). In addition, if the graph structure creates a scenario where many arcs are contending to write to a small set of states, serialization bottleneck may appear and the propagation technique could scale as line (c).

To minimize the synchronization cost for a given problem size, we need to choose the approach corresponding to the

lowest-lying line in Figure 5. For a small number of active states or arcs we should choose the propagation technique. For a larger number of arcs, however, the choice is highly dependent on the application graph structure and the write conflict resolution support in the underlying implementation platform.

> THE HIGHEST PERFORMING ALGORITHM STYLE VARIES WITH THE IMPLEMENTATION PLATFORM.

PORTABILITY AND PRODUCTIVITY IMPLICATIONS

The aggregation technique requires only standard global barriers for synchronization, whereas the propagation technique requires underlying write-conflict-resolution support, which can vary significantly across different platforms. Graph traversal implementation using the aggregation technique is more portable, while the one using the propagation technique may face portability issues if its architecture depends on one particular implementation of the write-conflict-resolution support.

The propagation approach is a more intuitive and productive technique for expressing the traversal process. It also often requires fewer lines of code, as it leverages the write conflict resolution support in the implementation platform. However, if code portability is a major concern when implementing the inference engine, the productivity tradeoffs of developing for multiple platforms are less clear.

TRANSITION EVALUATION GRANULARITY

One can also define two granularities for recognition network transition evaluation: evaluation based on states and evaluation based on arcs. In a parallel implementation, we must define units of work that can be done concurrently. State-based evaluation defines a unit of work as the evaluation of all outgoing or incoming arcs associated with a state. Arc-based evaluation defines a unit of work as the evaluation of a single arc.

The choice of evaluation granularity has direct implications on the efficiency of SIMD level processing, as each unit of work can be mapped onto a SIMD lane in a processor. High SIMD efficiency for a given SIMD width indicates that the algorithm is likely to benefit greatly from an even wider SIMD unit in future processors. We explore the implications of evaluation granularity on program control flow and data layout in this section.

CONTROL FLOW IMPLICATIONS

SIMD operations improve performance by executing the same operation on a set of data elements packed into a contiguous vector. Thus, SIMD efficiency is highly dependent on the ability of all lanes to synchronously execute useful instructions. When all lanes are fully utilized for an operation, we call the operation "synchronized." When operations are not synchronized, we consider them "divergent."

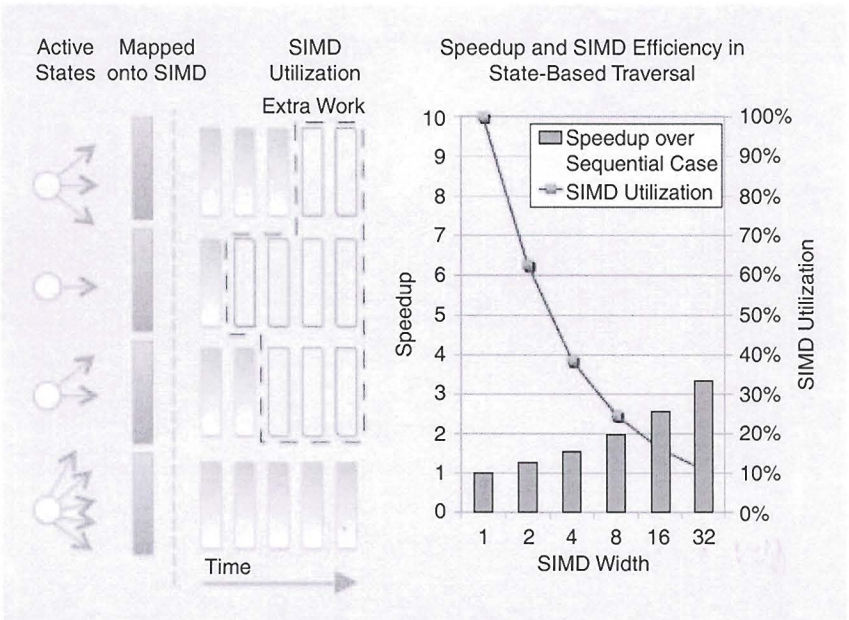
For the state-based approach, we see in Figure 6 that the control flow diverges as some lanes are idle, while others are conducting useful work. In our recognition network, the number of outgoing arcs of the active states ranges from one to 897. The bar chart in Figure 6 shows that the state-based evaluation granularity incurs significant penalties with increasing SIMD width. A 32-wide SIMD achieves only 10% utilization and achieves only a 3.3× speedup over a sequential version.

We can eliminate this control flow divergence by using the arc-based approach, as each arc evaluation presents a constant amount of work. However, such fully synchronized control flow requires extra instruction overhead, as well as extra storage overhead. For each arc evaluation to be an independent task, more tasks have to be defined and each arc must have a reference to its source state. We must manage more tasks and store more information for every arc we evaluate.

DATA LAYOUT IMPLICATIONS

Data accesses can be classified as "coalesced" or "uncoalesced." A "coalesced" memory access loads a consecutive vector of data that directly maps onto the SIMD lanes in the processing unit. Such accesses efficiently utilize the available memory bandwidth. "Uncoalesced" accesses, on the other hand, load nonconsecutive data elements to be processed by the vector units thereby wasting bandwidth.

During the traversal process, we access an arbitrary subset of nonconsecutive states or arcs in the recognition network in each



[FIG6] **SIMD unit utilization in the active state-based traversal.**

iteration resulting in uncoalesced memory accesses. One solution to this is to explicitly gather all required information into a temporary buffer such that all later accesses to the temporary buffer will be coalesced. This would help data coalescing at the expense of increasing the number of memory locations accessed in each iteration. This tradeoff is more sensitive in a cache-based architecture, since the enlarged working set size leads to capacity misses in the cache.

> FOR THE STATE-BASED APPROACH, WE SEE IN FIGURE 6 THAT THE CONTROL FLOW DIVERGES AS SOME LANES ARE IDLE, WHILE OTHERS ARE CONDUCTING USEFUL WORK.

IMPLEMENTATION OF THE INFERENCE ENGINE

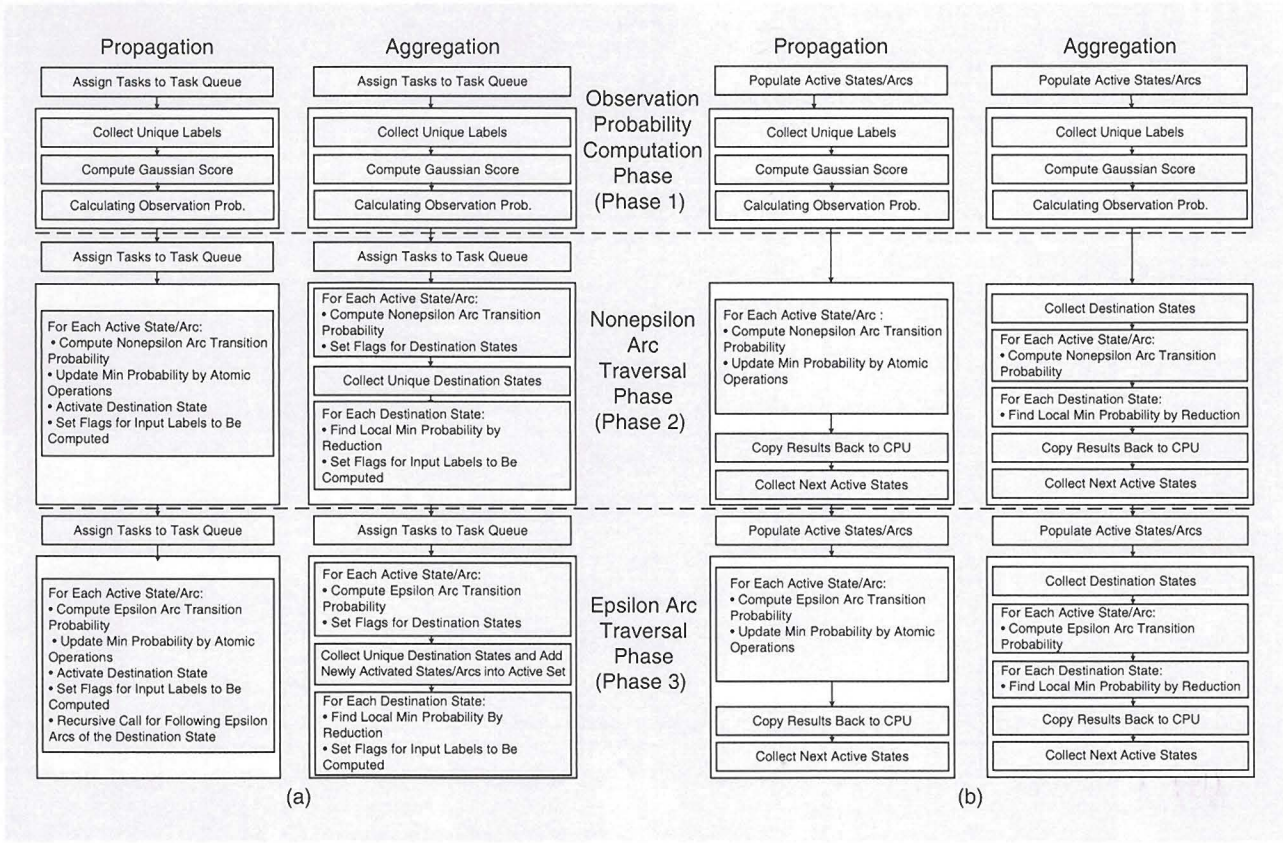
We examine full implementations of the inference engine on two separate platforms: 1) an Intel Core i7 multicore processor and 2) a NVIDIA GTX280 manycore processor. We discuss the control flow and data structure implementations as well as core-level load balancing issues and recognition network optimization.

CONTROL FLOW DESIGN

The flow diagrams in Figure 7(a) and (b) describe the control flow of our implementations in each iteration of the inference engine. We present the multicore implementation in Figure 7(a) and the manycore implementation in Figure 7(b). Both implementations are illustrated with two flow diagrams, one for graph traversal by propagation and one for traversal by aggregation. All of the flow diagrams have the three distinct execution phases as defined in the section "Characteristics of LVCSR." Each phase in the algorithm can involve multiple steps, illustrated as boxes in the flow diagram.

Each step in the flow chart represents an algorithmic step that depends on the results from the previous step, where the dependency is respected by applying a global barrier at the end of every step. For example, Phase 1 involves three steps: collecting the unique labels to extract the context-dependent state models to be computed, computing the Gaussian scores with the input features and Gaussian parameters referenced by the models, and finally calculating the observation probability of the models utilizing their mixture weights and corresponding Gaussian scores. In Phases 2 and 3, the result of each step is written to memory and the next step reads it from memory, usually in a different order. With respect to each core on a multicore/manycore chip, intermediate results are communicated between the cores across the different algorithmic steps through memory.

The distinction between evaluation by state and evaluation by arc is also illustrated in Figure 7. In the Core i7 implementations in Figure 7(a), the task queues created at the beginning of Phase 2 and 3 can involve tasks based on states or arcs, and the following steps are performed according to the granularity of



[FIG7] Flow diagram of the algorithm styles explored on both the Core i7 and the GTX280 platforms. (a) Core i7 implementation. (b) GTX280 implementation.

tasks in the task queue. In the GTX280 implementation in Figure 7(b), the run-time data structures are populated with either active states or active arcs at the beginning of Phases 1 and 3, and the following steps in Phases 2 and 3 are performed according to the data format in the run-time data structures. The data structure population of Phase 2 is done before Phase 1 as an optimization to allow the unique set of labels to be extracted at the same time to avoid duplications in the expensive Gaussian score computation.

> SIMD OPERATIONS IMPROVE PERFORMANCE BY EXECUTING THE SAME OPERATION ON A SET OF DATA ELEMENTS PACKED INTO A CONTIGUOUS VECTOR.

DATA STRUCTURE IMPLEMENTATIONS

For the Core i7 implementation, all data structures are stored in main memory and the data working set is transparently managed by the hardware cache hierarchy. To utilize the cache more efficiently, all the outgoing arcs information from a source state is stored consecutively in main memory. Since, in the state-based traversal, the outgoing arcs from the same source state are processed successively in the same thread of execution, this layout reduces the memory access time of Phases 2 and 3.

For the GTX280 implementation, there are two levels of memory hierarchy for the graphics processing unit (GPU) with orders-of-magnitude differences in throughput. Data in the main memory on the host system can be accessed at 2.5 GB/s from the GPU. Data in device memory on the GPU board can be accessed at 120 GB/s from the GPU. Phases 2 and 3 implement graph traversal functions that are memory access intensive. It is essential to keep the working set in device memory for high bandwidth access. The GTX280 provides 1 GB of device memory on the GPU board that can fit the acoustic model (130 MB), the language model (400 MB), and various temporary graph traversal data structures. We architect all graph traversal steps to run exclusively on the GPU with intermediate results stored in the device memory. This avoids the host-device memory transfer bottleneck and allows the Compute Unified Device Architecture (CUDA) kernels to utilize 20–120 GB/s memory bandwidth. However, some steps such as prefix scan incur significant penalty when parallelized, requiring more total operations to reach the same results. This is reflected in the lower overall speedup for Phases 2 and 3. Not all intermediate data can fit in the device memory, however. The traversal history data is copied back to the host system at regular intervals to save space. Since history data is only used at the very end of the traversal process, the data transfer is a one-way, device-to-host copy. This transfer involves around 10 MB of data/s, which translates to less than 5 ms of transfer time on a channel with 2.5 GB/s bandwidth, and is accounted for in the sequential overhead measurements.

CORE-LEVEL LOAD BALANCING

Core-level load imbalance is a key factor that limits parallel speedup in the inference engine for LVCSR [11]. Load imbalance occurs when the states in the recognition network are statically assigned to each core and the working set migrates every iteration depending on input audio features. Load imbalance can be eliminated by dynamically assigning work to idle cores in each iteration.

For the Core i7 implementation, we use a distributed task queue programming framework [17]. The distributed task queue defines a task as a function that executes in one thread and can be scheduled as a unit. The programmer describes an array of tasks for arc or state computation and the framework monitors for idle cores and load balances the system during run time. Load balancing is done as follows: the distributed task queue manages one physical queue per thread, assigning each thread a "preferred" queue that it accesses with highest priority. Before each phase in the inference engine implementation, the tasks are explicitly and evenly distributed among the set of task queues. Each thread processes tasks in its respective local queue. When a thread becomes idle, the task queue steals a task for the idle thread from a nonempty queue in another thread, thereby load balancing the system. This lazy load-balance policy adds minimal overhead during execution and frees the programmer from concerns of core-level load balancing.

For the GTX280 implementation, we use the CUDA programming framework [18], which provides the key abstractions for data parallel programming. We assert control over a hierarchy of thread groups by programming with conventional C code for one thread. The CUDA framework then constructs the necessary SIMD instructions in a thread group at compile time, and distributes and load balances thread groups in hardware at run time onto the many cores of the GTX280.

RECOGNITION NETWORK OPTIMIZATION

In the epsilon arc traversal, some states in the recognition network can reach destination states through multiple levels of expansion through the epsilon arcs. For example, State 5 in Figure 3 reaches States 8, 9, and 11 through its epsilon arc transitions. Our recognition network has chains of epsilon arcs that are up to four levels deep.

In the propagation approach of the Core i7 implementation, the traversal over epsilon arcs is done recursively over the epsilon network. Multiple levels of states connected by the epsilon arcs are updated in each recursive step. However, this kind of recursive traversal does not work well for the aggregation approach of the Core i7 implementation or the data parallel GTX280 implementations. This is because a parallel expansion over one level of epsilon arcs requires a global barrier of synchronization limiting the amount of parallelism.

We insert look-ahead epsilon arcs into the network so that every state connected by multiple expansions of epsilon arcs is reachable in only one level. For example in Figure 3, we insert an epsilon arc between State 5 and State 11, such that State 11 is reachable within one level of epsilon arc expansion from State 5. For our recognition network, this type of insertions increased the total number of arcs in the recognition network

[TABLE 1] ACCURACY IN WER FOR VARIOUS BEAM SIZES AND CORRESPONDING DECODING SPEED IN RTF.

AVERAGE NUMBER OF ACTIVE STATES		32,820	20,000	10,139	3,518
	WER	41.6	41.8	42.2	44.5
	SEQUENTIAL	4.36	3.17	2.29	1.20
RTF	MULTICORE	1.23	0.93	0.70	0.39
	MANYCORE	0.40	0.30	0.23	0.18

by only 2.0%. The number of arcs traversed during decoding is increased by 1.7%, while total latency spent in graph traversal is reduced by 19%. This increase in network size is negligible compared to the significant savings from eliminating potential multilevel epsilon arc expansion overhead.

EVALUATION OF THE INFERENCE ENGINE

SPEECH MODELS AND TEST SETS

The speech models are taken from the SRI CALO real-time meeting recognition system [19]. The front end uses 13 dimensional perceptual linear prediction (PLP) features with first, second, and third order differences, is vocal tract-length normalized, and is projected to 39 dimensions using heteroscedastic linear discriminant analysis (HLDA). The acoustic model is trained on conversational telephone and meeting speech corpora using the discriminative minimum phone error (MPE) criterion. The language model is trained on meeting transcripts, conversational telephone speech, and Web and broadcast data [20]. The acoustic model includes 52,000 triphone states that are clustered into 2,613 mixtures of 128 Gaussian components.

The pronunciation model contains 59,000 words with a total of 80,000 pronunciations. We use a small backoff bigram language model with 167,000 bigram transitions. The recognition network is an $H \circ C \circ L \circ G$ model compiled using WFST techniques and contains 4.1 million states and 9.8 million arcs.

The test set consisted of excerpts from NIST conference meetings taken from the "individual head-mounted microphone" condition of the 2007 NIST Rich Transcription evaluation. The segmented audio files total 44 mins in length and comprise ten speakers. For the experiment, we assumed that the feature extraction is performed offline so that the inference engine can directly access the feature files. The meeting recognition task is very challenging due to the spontaneous nature of the speech. The ambiguities in the sentences require larger number of active states to keep track of alternative interpretations which leads to slower recognition speed.

Our recognizer uses an adaptive heuristic to adjust the search beam size based on the number of active states. It controls the number of active states to be below a threshold to guarantee that all traversal data fits within a pre-allocated memory space. Table 1 shows the decoding accuracy, in terms of WER with varying thresholds and the corresponding decoding speed on various platforms. The recognition speed is represented by the real-time factor (RTF) that is computed as the total decoding time divided by the duration of the input speech.

As shown in Table 1, the multicore and manycore implementations can achieve significant speedup for the same number of active states. More importantly, for the same RTF, parallel implementations provide a higher recognition accuracy. For an RTF of 1.2, WER reduces from 44.5 to 41.6% going from a sequential to a multicore implementation. For an RTF of 0.4, WER reduces from 44.5 to 41.6% going from a multicore implementation to a manycore implementation.

For the experiments in the next few sections, we choose a beam-width setting that maintains an average of 20,000 active states to analyze the performance implications in detail. All algorithm styles and the sequential implementation are functionally equivalent with negligible differences in decoding output.

[TABLE 2] PARAMETERS FOR THE EXPERIMENTAL PLATFORMS.

TYPE	MULTICORE	MANYCORE
PROCESSOR	CORE I7 920	GTX280 (+CORE2 Q9550)
CORES	FOUR CORES (SMT)	30 CORES
SIMD WIDTH	FOUR LANES	EIGHT PHYSICAL, 32 LOGICAL
CLOCK SPEED	2.66 GHZ	1.296 GHZ
SP GFLOP/S	85.1	933
MEMORY CAPACITY	6 GB	1 GB (8 GB)
MEMORY BW	32.0 GB/s	141.7 GB/s
COMPILER	ICC 10.1.015	NVCC 2.2

EXPERIMENTAL PLATFORM SETUP

The specifications of the experimental platforms are listed in Table 2. The peak value of the single precision giga floating point operations per second (SP GFLOPS/s) and the memory bandwidth are the theoretical bounds. For the manycore platform setup, we use a Core2 Quad-based host system with 8 GB host memory and a GTX280 graphics card with 1 GB of device memory.

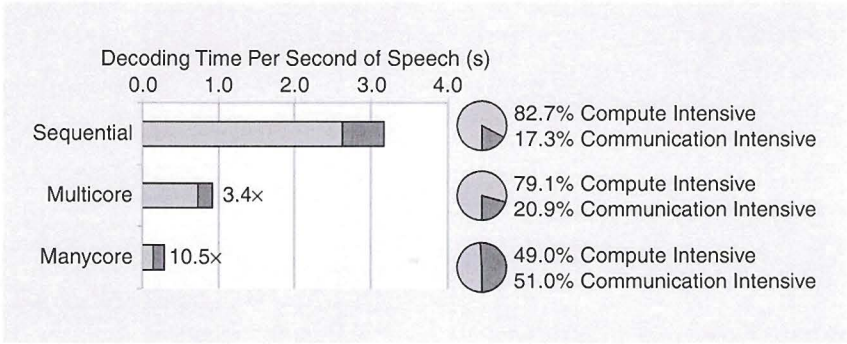
OVERALL PERFORMANCE

We analyze the performance of our inference engine implementations on both the Core i7 multicore processor and the GTX280 manycore processor. The sequential baseline is implemented on a single core in a Core i7 quadcore processor. It utilizes a SIMD-optimized Phase 1 routine and non-SIMD graph traversal routine for Phases 2 and 3. This configuration is chosen to show the best performance of sequential baseline as explained in the section, "SIMD Utilization Efficiency Evaluation." When comparing to this highly optimized sequential baseline implementation, we achieve 3.4× speedup using all cores of Core i7 and 10.5× speedup on GTX280.

The performance gain is best illustrated in Figure 8 by highlighting the distinction between the compute-intensive phase (green bar) and the communciation intensive phase (pink bar). The compute-intensive phase achieves 3.6× speedup on the multicore processor and 17.7× on the manycore processor, while the communication-intensive phase

achieves only 2.8× speedup on the multicore processor and 3.7× on the manycore processor.

The speedup numbers indicate that synchronization overhead dominates the run time as more processors need to be coordinated in the communication intensive phase. In terms of the ratio between the compute and communication-intensive phases, the pie charts in Figure 8 show that 82.7% of the time in the sequential implementation is spent in the compute-intensive phase of the application. As we scale to the manycore implementation, the compute-intensive phase becomes proportionally less dominant, taking only 49% of the total run time. The increasing dominance of the communication-intensive phase motivates a detailed examination of the parallelization implications in the communication-intensive phases of our inference engine.



[FIG8] **Ratio of computation-intensive phase of the algorithm versus communication intensive phase of the algorithm.**

DETAILED PERFORMANCE ANALYSIS

We present a detailed analysis of the various algorithm styles proposed in the section "Algorithm Styles of the Inference Engine" that implement the graph traversal process in the communication-intensive phase of our inference engine. The run time performances of the different algorithm styles are summarized in Table 3, where each column represents a different implementation, and each row provides performance and speedup numbers for the implementations. We found that the sequential overhead in our implementation is less than 2.5% of the total run time even for the fastest implementation. This demonstrates that we have a scalable software architecture that promises greater potential speedups with more platform parallelism expected in future generations of processors.

We also find that the fastest algorithm style differs for each platform. Table 3 shows the fastest algorithm style for each platform. For Core i7 the the fastest algorithm style is propagation-by-states and for GTX280 the fastest style is propagation-by-arcs. We evaluate these results with respect to the synchronization cost and SIMD utilization efficiency in this section.

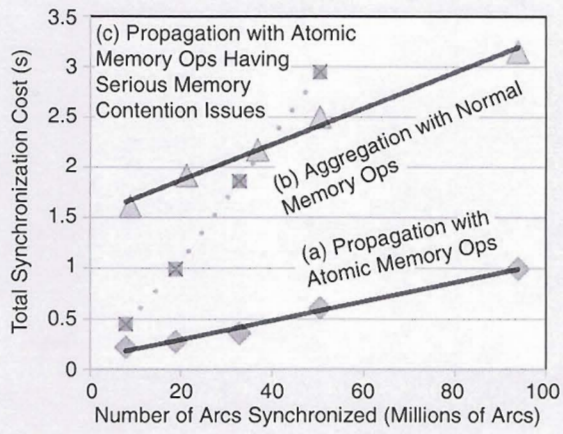
SYNCHRONIZATION COST EVALUATION

Synchronization cost for the destination state updates in the graph traversal differs significantly between the aggregation and propagation algorithm styles. As shown in Table 3, on both the multicore and manycore platforms, aggregation-based implementations achieved worse performance for the communication-intensive phase compared to the sequential implementation. This illustrates the scenario where the cost of parallel coordination overwhelmed the benefit of parallel execution. The high cost of parallel coordination can be explained by Figure 7. Phases 2 and 3 of the aggregation techniques have two extra steps in both the multicore and the manycore implementations, adding to the overhead of managing graph traversal.

The propagation algorithm style, however, is able to outperform the sequential implementation for the communication-intensive phase. As an example, we show the observed scaling characteristics for the synchronization cost on the manycore processor in Figure 9. This figure uses problem sizes shown in Table 1. The X-axis shows the number of active arcs evaluated and the Y-axis shows the execution time of synchronization. Line (a) in Figure 9 corresponds to the propagation-by-arcs style. It has a lower overall synchronization cost compared to line (b), which is from the aggregation-by-arcs style. The low synchronization cost in the propagation-by-arcs style is the result of careful tuning of the algorithm to avoid writing to shared memory locations in device memory by every task. Writing to shared memory locations in device memory by every

[TABLE 3] RECOGNITION PERFORMANCE NORMALIZED FOR 1 S OF SPEECH FOR DIFFERENT ALGORITHM STYLES. SPEEDUP REPORTED OVER OPTIMIZED SEQUENTIAL VERSION OF THE PROPAGATION-BY-STATES STYLE.

	CORE i7	CORE i7			GTX280			
SECONDS (%)	SEQUENTIAL PROP. BY STATES	PROP. BY STATES	PROP. BY ARCS	AGGR. BY STATES	PROP. BY STATES	PROP. BY ARCS	AGGR. BY STATES	AGGR. BY ARCS
PHASE 1	2.623 (83%)	**0.732 (79%)**	0.737 (73%)	0.754 (29%)	0.148 (19%)	**0.148 (49%)**	0.147 (12%)	0.148 (16%)
PHASE 2	0.474 (15%)	**0.157 (17%)**	0.242 (24%)	1.356 (52%)	0.512 (66%)	**0.103 (34%)**	0.770 (64%)	0.469 (51%)
PHASE 3	0.073 (2%)	**0.035 (4%)**	0.026 (3%)	0.482 (19%)	0.108 (15%)	**0.043 (14%)**	0.272 (23%)	0.281 (31%)
SEQUENTIAL OVERHEAD	—	**0.001**	0.001	0.001	0.008 (1.0%)	**0.008 (2.5%)**	0.014 (1.2%)	0.014 (1.6%)
TOTAL	3.171	**0.925**	1.007	2.593	0.776	**0.301**	1.203	0.912
SPEEDUP	1	**3.43**	3.15	1.22	4.08	**10.53**	2.64	3.48



[FIG9] **Synchronization cost for GTX280 as it scales over various number of arcs to traverse.**

task creates memory contention and serialization bottlenecks resulting in the significant increase in synchronization penalty shown as line (c). This illustrates that although the propagation technique can have significant lower synchronization cost than the aggregation technique, synchronization bottlenecks due to graph structure-induced memory access contentions can still significantly degrade performance in a highly parallel design.

SIMD UTILIZATION EFFICIENCY EVALUATION

SIMD utilization efficiency is increasingly important in multicore and manycore programming. Neglecting SIMD or a poor utilization of SIMD can lead to an order of magnitude degradation in performance. The control for SIMD utilization efficiency lies in the granularity of tasks assigned to the SIMD lanes. In the case with the manycore propagate-based implementation, the communication-intensive phase achieved a 4× boost in performance in switching from a state-based task granularity to an arc-based task granularity. One key step, "compute nonepsilon arc transition probability," was accelerated by more than 9× when making this switch.

In the multicore implementation, the performance tradeoff for task granularity is more complex. The overhead of managing arc-based task granularity does not only include creating finer grained tasks, but it also results in various cache implications such as increased capacity misses caused by maintaining a larger working set and using less regular data access patterns. Although this overhead can be partially compensated by more efficient SIMD execution, applying SIMD for the arc traversal (Phases 2 and 3) does not yield any speedup in the Core i7 implementations, since the overhead of gathering the data exceeds the speedup achievable by a relatively narrow four-wide SIMD. Thus, the arc traversal steps still perform faster with the state-based task granularity. For this reason, we did not evaluate the aggregation-by-arcs style in Core i7 implementations.

CONCLUSIONS

In this article, we exposed the fine-grained application concurrency in a HMM-based inference engine for LVCSR and optimized a parallel software architecture for the inference process with less than 2.5% sequential run time overhead, promising significant potential for further speedup on future parallel platforms.

We explored two important aspects of the algorithmic level design space for parallel scalability to account for different support and efficiency of concurrent task synchronization and SIMD utilization on multicore and manycore platforms. While we achieved significant speedups compared to highly optimized sequential implementation: 3.4× on an Intel Core i7 multicore processor and 10.5× on a GTX280 NVIDIA manycore processor, the fastest algorithm style differed for each platform. Application developers must take into account underlying hardware architecture features such as synchronization operations and the SIMD width when they design algorithms for parallel platforms.

Automatic speech recognition is a key technology for enabling rich human-computer interaction in emerging applications. Parallelizing the implementation is crucial to reduce recognition latency, increase recognition accuracy, enabling the handling of more complex language models under time constrains. We expect that an efficient speech recognition engine will be a component in many exciting new applications to come.

ACKNOWLEDGMENTS

The authors would like to thank Pradeep Dubey, Lynda Grindstaff, and Yasser Rasheed at Intel for initiating and supporting this research and Nelson Morgan, Andreas Stolcke, and Adam Janin at ICSI for insightful discussions and continued support in the infrastructure used in this research. The authors also thank NVIDIA for donating the hardware used.

This research is supported in part by the Ministry of Education, Science and Technology, Republic of Korea under the Brain Korea 21 Project, the Human Resource Development Project for IT SoC Architect, and by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD KRF-2007-357-D00228). It is also supported in part by Microsoft (Award 024263), Intel (Award 024894) funding, matching funding by U.C. Discovery (Award DIG07-10227), and by an Intel Ph.D. research fellowship.

AUTHORS

Kisun You (kisun.you@ieee.org) received his B.S. degree in electrical engineering and computer science from Seoul National University, Korea, in 2002, where he is currently a Ph.D. candidate. His research interests include analysis and optimization of speech recognition for manycore and multicore platforms and its efficient hardware design. He conducted research as an intern at Intel Application Research Labs in 2008. He is a Student Member of the IEEE.

Jike Chong (jike@chongjike.net) received his B.S. and M.S. degrees from Carnegie Mellon University, Pittsburgh, Pennsylvania in 2001. He is currently a Ph.D. candidate at University of California, Berkeley, working on application frameworks in speech recognition and computational finance to help domain experts efficiently utilize highly parallel computation platforms. Previously, he worked for Sun Microsystems, Inc., designing microarchitecture features for highly parallel processors. He also conducted research at Intel Application Research Labs and Xilinx Research Labs. He is an Intel Ph.D. research fellow, and a Student Member of Eta Kappa Nu, Tau Beta Pi, and the IEEE.

Youngmin Yi (ymyi76@gmail.com) received his B.S. and Ph.D. degrees from Seoul National University, Korea in 2000 and 2007, respectively. He is currently a postdoctoral researcher at the University of California, Berkeley. His research interests include parallel software system design methodology, frameworks for efficient and productive parallel software designs, developing parallel applications for machine learning, and studying performance implications of the applications in manycore architectures. He is a Member of the IEEE.

Ekaterina Gonina (egonina@eecs.berkeley.edu) received a B.S. degree in computer science from the University of Illinois, Urbana-Champaign in 2008. She is currently pursuing a Ph.D. degree in computer science at the University of California, Berkeley. Her research interests include parallel application development, analysis and optimization on manycore and multicore platforms, and implications of various computational loads on the architecture of parallel platforms.

Christopher J. Hughes (christopher.j.hughes@intel.com) received his Ph.D. degree from the University of Illinois at Urbana-Champaign in 2003. He is currently a staff researcher at Intel Labs in the Throughput Computing Lab. His research interests are emerging workloads and computer architectures. His recent work focuses on mapping computationally intensive applications to next-generation multicore and manycore CPUs and GPUs. He is a Member of the IEEE.

Yen-Kuang Chen (y.k.chen@ieee.org) received the B.S. degree from National Taiwan University and the Ph.D. degree from Princeton University in New Jersey. He is a principal engineer at Intel Labs. His research interests include developing innovative multimedia applications, studying the performance bottleneck in current architectures, and designing next-generation microprocessors/platforms. He is a Senior Member of the IEEE.

Wonyong Sung (wysung@snu.ac.kr) received the Ph.D. degree in electrical and computer engineering from the University of California, Santa Barbara, in 1987. He has been a faculty member at Seoul National University since 1989. His major research interests are the development of fixed-point optimization tools, implementation of VLSI for digital signal processing, and development of parallel processing software for signal processing. He has been a design and implementation technical committee member of the IEEE Signal Processing Society since 1999. He is a Senior Member of the IEEE.

Kurt Keutzer (keutzer@eecs.berkeley.edu) received his B.S. degree in mathematics from Maharishi International University in 1978 and his M.S. and Ph.D. degrees in computer science from Indiana University in 1981 and 1984, respectively. He joined AT&T Bell Laboratories in 1984, and Synopsys, Inc. in 1991, where he became chief technical officer and senior vice-president of research. He became a professor of electrical engineering and computer science at the University of California, Berkeley in 1998, and served as the associate director of the Gigascale Silicon Research Center. He cofounded the Universal Parallel Computing Research Center at Berkeley in 2007. He is a Fellow of the IEEE.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Dept., Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[2] A. Obukhov and A. Kharlamov, "Discrete cosine transform for 8 × 8 blocks with CUDA," NVIDIA White Paper, Oct. 2008.

[3] V. Podlozhnyuk, "FFT-based 2D convolution," NVIDIA White Paper, June 2007.

[4] H. Ney and S. Ortmanns, "Dynamic programming search for continuous speech recognition," *IEEE Signal Processing Mag.*, vol. 16, no. 5, pp. 64–83, 1999.

[5] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Process. Lett.*, vol. 17, no. 1, pp. 5–20, 2007.

[6] A. Janin, "Speech recognition on vector architectures," Ph.D. dissertation, Univ. California, Berkeley, CA, 2004.

[7] M. Ravishankar, "Parallel implementation of fast beam search for speaker-independent continuous speech recognition," Comput. Sci. Automat., Indian Inst. Sci., Bangalore, India, Tech. Rep., 1993.

[8] K. Agaram, S. W. Keckler, and D. Burger, "A characterization of speech recognition on modern computer systems," in *Proc. IEEE Int. Workshop Workload Characterization (WWC-4)*, 2001, pp. 45–53.

[9] S. Ishikawa, K. Yamabana, R. Isotani, and A. Okumura, "Parallel LVCSR algorithm for cellphone-oriented multicore processors," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, Toulouse, France, 2006, pp. 177–180.

[10] K. You, Y. Lee, and W. Sung, "OpenMP-based parallel implementation of a continuous speech recognizer on a multi-core system," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, Taipei, Taiwan, 2009, pp. 621–624.

[11] S. Phillips and A. Rogers, "Parallel speech recognition," *Int. J. Parallel Program.*, vol. 27, no. 4, pp. 257–288, 1999.

[12] P. R. Dixon, T. Oonishi, and S. Furui, "Fast acoustic computations using graphics processors," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, Taipei, Taiwan, 2009, pp. 4321–4324.

[13] P. Cardinal, P. Dumouchel, G. Boulianne, and M. Comeau, "GPU accelerated acoustic likelihood computations," in *Proc. Interspeech*, 2008, pp. 964–967.

[14] J. Chong, Y. Yi, N. R. Satish, A. Faria, and K. Keutzer, "Data-parallel large vocabulary continuous speech recognition on graphics processors," in *Proc. Int. Workshop Emerging Applications and Manycore Architectures*, 2008, pp. 23–35.

[15] M. Mohri, F. Pereira, and M. Riley, "Weighted finite state transducers in speech recognition," *Comput. Speech Lang.*, vol. 16, no. 1, pp. 69–88, 2002.

[16] S. Kanthak, H. Ney, M. Riley, and M. Mohri, "A comparison of two LVR search optimization techniques," in *Proc. Int. Conf. Spoken Language Processing (ICSLP)*, Denver, CO, 2002, pp. 1309–1312.

[17] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proc. Int. Symp. Computer Architecture (ISCA)*, 2007, pp. 162–173.

[18] NVIDIA Corp., *NVIDIA CUDA Programming Guide, Version 2.2 beta*, Mar. 2009.

[19] G. Tur, A. Stolcke, L. Voss, J. Dowding, B. Favre, R. Fernandez, M. Frampton, M. Frandsen, C. Frederickson, M. Graciarena, D. Hakkani-Tür, D. Kintzing, K. Leveque, S. Mason, J. Niekrasz, S. Peters, M. Purver, K. Riedhammer, E. Shriberg, J. Tien, D. Vergyri, and F. Yang, "The CALO meeting speech recognition and understanding system," in *Proc. IEEE Spoken Language Technology Workshop*, 2008, pp. 69–72.

[20] A. Stolcke, X. Anguera, K. Boakye, O. Cetin, A. Janin, M. Magimai-Doss, C. Wooters, and J. Zheng, "The SRI-ICSI spring 2007 meeting and lecture recognition system," *Lect. Notes Comput. Sci.*, vol. 4625, no. 2, pp. 450–463, 2008. SP