1

Scalable Parallelization of Automatic Speech Recognition

Jike Chong, Ekaterina Gonina, Kisun You, Kurt Keutzer

1.1 Introduction

Automatic speech recognition (ASR) allows multimedia content to be transcribed from acoustic waveforms into word sequences. It is an exemplar of a class of machine learning applications where increasing compute capability is enabling new industries such as automatic speech analytics. Automatic speech analytics help customer service call centers search through recorded content, track service quality, and provide early detection of service issues. Fast and efficient ASR enables economic employment of a plethora of text-based data analytics on multimedia contents, opening the door to many possibilities.

In this chapter, we describe our approach approach for scalable parallelization of the most challenging component of ASR: the speech inference engine. This component accesses a large irregular data structure with millions of states and arcs representing a human speech model. Parallel implementation of the inference engine requires frequent synchronizations, and has unpredictable data working set and memory access patterns because of direct dependency on the audio input.

We demonstrate that parallelizing an application is much more than just recoding the program in another language. It requires careful consideration of a series of concerns to successfully exploit the full parallelization potential of an application. Using our approach, we were able to achieve more than 3.4x speed up on an Intel Core i7 multicore processor and more than 11x speedup on an NVIDIA GTX280 manycore processor for the ASR inference engine. This performance improvement opens up many opportunities for latency-limited as well as throughput-limited applications of automatic speech recognition.

1.1.1 Automatic Speech Recognition

Recognition of human speech is a complex task, especially considering the significant variation in the voice quality, speed, and pronunciation among different speakers. Furthermore, differences among languages as well as the speech recording environments pose further challenges to an effective recognition system. After decades of scientific research, many researchers have converged on the hidden Markov model (HMM) "extract and inference" system as a standard setup. In this setup, the acoustic signal is treated as the observed sequence of events and the sentences to be recognized are considered the "hidden cause" of the acoustic signal. In this chapter, we focus on this setup and discuss approaches to speedup the inference process on both multicore and manycore parallel computation platforms.

As shown in Figure 1.1, ASR first extracts representative features from an input waveform and then decodes the feature sequence to produce a word sequence. The feature extraction process involves a sequence of signal processing steps. It aims

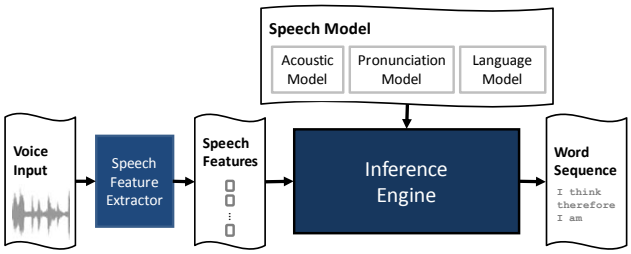

Figure 1.1 System architecture of a large vocabulary continuous speech recognition application

to minimize variation among speaker voice quality and room acoustics and preserve features most useful to distinguishing word sequences. The features are then used in an inference process, where the feature sequence is iteratively compared to a probabilistic speech model. The speech model contains an acoustic component, a pronunciation component, and a language component. Both the acoustic and language components are trained off-line using a set of powerful statistical learning techniques. The acoustic model is often represented as a multi-component Gaussian mixture model, which takes into account slight differences in the pronunciations of a phoneme[1]. The pronunciation model is a dictionary of word pronunciations. Finally, the language model provides the likelihood of appearances of words and phases in the language.

To recognize a different language in a different environment, only the speech model and the feature extractor need to be changed. The inference engine stays unchanged producing the most-likely interpretation of the input sequence by traversing the speech model for the specified language.

1.1.2 Our Methodology

In this chapter, we illustrate our approach for scalable parallelization of a speech inference engine. This process can also be applied to other similar applications in machine learning such as machine translation. Our approach involves a judicious design of a software architecture to efficiently exploit concurrency in an application. We define a software architecture as a hierarchical composition of *patterns*, which are problem-solution pairs to recurring problems that experts in a problem domain gradually learn and "take for granted" (Keutzer and Mattson (2009)). Software architecture is based on a careful analysis of the concurrency sources in the given application.

[1] An abstract unit of sound in the phonetic system of a language

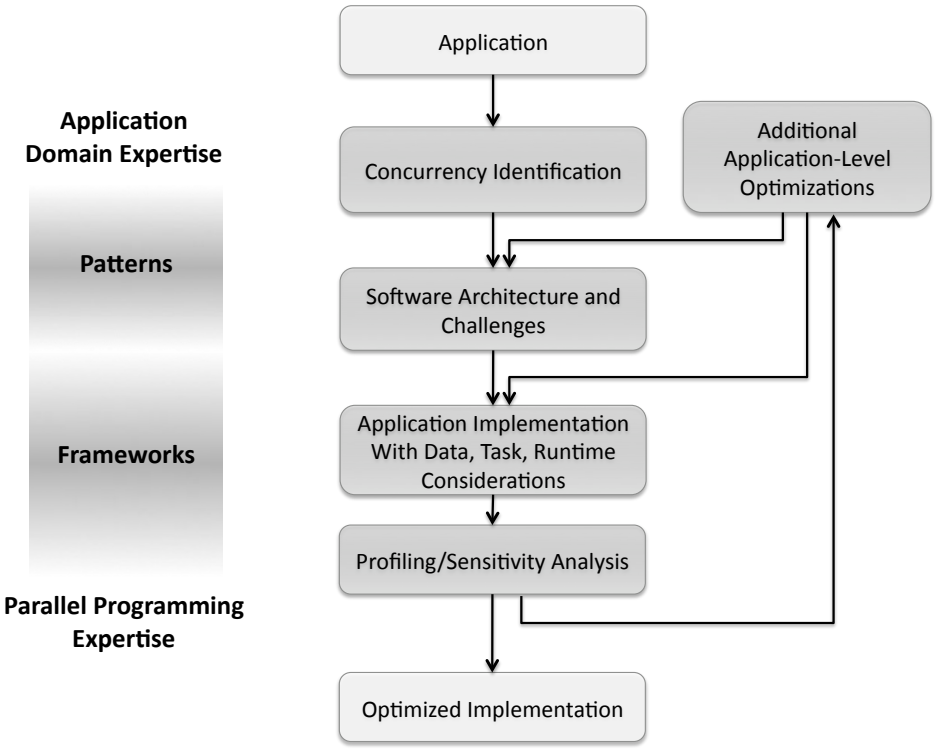

Figure 1.2 Steps in parallelizing an application on a parallel platform

We illustrate a step-by-step parallelization process in Figure 1.2 and describe the benefits of our approach. Each section of the chapter corresponds to a step in the parallelization process as following:

Concurrency Identification: identify a rich source of concurrency that improves application-specific performance metric to obtain continued parallel scalability. (Section 1.2)

Software Architecture and Challenges: construct a software architecture with software design patterns and use the pattern descriptions to help identify challenges when implementing the application (Section 1.3)

Application Implementation: take care of data, task and runtime efficiency concerns in an implementation for a specific parallel platform. (Section 1.4)

Profiling/Sensitivity Analysis: analyze the performance by evaluating particular application-specific performance metrics and evaluate sensitivity to changes in implementation styles.(Section 1.5)

Additional Application-level Optimizations: Examine and apply application-

level transformations to mitigate performance bottlenecks that cannot be removed by application-unaware optimizations (Section 1.6)

We conclude and summarize our key lessons learned in Section 1.7, and highlight the needs for patterns and frameworks for productive parallel programming.

1.2 Concurrency Identification

Concurrency identification is the first and the most important step during the application parallelization process. Concurrency is a property of the application where application modules are identified to be independent. We can perform computation on the independent modules simultaneously and still produce logically correct results. During the parallelization process we exploit the concurrency in an application and maps it onto execution resources. Not all sources of concurrency in an application need to be exploited to achieve efficient execution on a parallel platform.

There are many sources of concurrency in a HMM based automatic speech recognition. These concurrency opportunities are often obvious for domain experts working in the application area. The challenge is to clarify the scalability and benefits of various sources of concurrency and evaluate which ones to exploit during parallelization. First, we enumerate these concurrency opportunities in ASR.

1. **Concurrency across sentences**: each sentence can be recognized independently.
2. **Concurrency across phases of the algorithm**: different phases of computation can be pipelined to handle multiple timesteps of data at the same time.
3. **Concurrency across acoustic model computation**: each input feature is compared to thousands of acoustic model elements (or phone models) at a time, the comparison with each phone model is independent.
4. **Concurrency across alternative interpretations of a sentence**: many thousands of alternative interpretations are maintained to avoid allowing local noise to cause elimination of the overall most likely interpretation.

Next, we outline the application specific performance metrics. There are three main performance metrics for ASR: improving accuracy, throughput, and latency (Chong et al. (2010)). Improving recognition accuracy opens up new domains of applications where there is less tolerance for recognition error; improving throughput lowers the cost of existing batch processing usage scenarios, and improving latency can allow more complex processing steps such as language translation to be integrated while still meeting real time latency constraints.

Table 1.1 illustrates the concurrency identification process for ASR. The table provides a comparison of the different levels of functional concurrency, as well as highlights the benefits in key application domain concerns.

Table 1.1 *Available application concurrency and the key application domain concerns being addressed*

Concurrency Sources	Concurrency Scalability (# ways)	Improving Accuracy	Improving Throughput	Improving Latency
Across sentences	300-500	Yes*	Yes	No
Across algorithm phases	3-5	Yes*	Yes	No
Across acoustic model computations	500-3000	Yes*	Yes	Yes
Across alternative interpretations	10000+	Yes*	Yes	Yes

* Speedup can be used to improve accuracy when the usage scenario is compute-capacity constrained, and the speedup enables more complex processing to be done within the same cost constraints

The **concurrency across sentences** is a popular concurrency source to exploit in deploying to todays computing clusters. It is the de facto parallelization approach for speech recognition researchers and developers alike. Each sentence is considered a separate task to be transcribed independently on a cluster node. Assuming an average sentence is approximately 10 seconds long, to transcribe a 60-minute talk, or 3600 seconds of speech, we would expect 300-500 way concurrency, which can be mapped to any number of computers in a cluster. However, this approach does not help in improving the latency of recognizing one particular sentence.

The **concurrency across algorithm phases** involves pipelining the algorithmic phases. For example, Ishikawa et al. (2006) explored this coarse-grained concurrency in implementing a pipeline of tasks on a cellphone-oriented multicore architecture. Although some speedup can be obtained using this approach, it involves significant effort to re-factor an implementation to target every new generation of parallel hardware. The source of concurrency is limited in scalability and exploiting it with pipelining does not improve recognition latency.

The **concurrency in acoustic model computation** involves estimating the likelihood of an input feature vector matching to particular units (or phones) in an acoustic model. There exists a 500-3000 way concurrency concentrated in a simple Gaussian mixture model computation kernel, representing up to 80% of the total computation time in the inference engine. Research by Dixon et al. (2007, 2009) and Cardinal et al. (2008) focused on speeding up this part of ASR on manycore accelerators. Both demonstrated moderate (approximately 5x) speedups using manycore accelerators and managed the alternative interpretations in the Viterbi search process on the host system. However, this approach introduces significant

overhead in copying intermediate results between the host and the manycore accelerator subsystem, which diminishes the benefits of potential speedup.

The **concurrency across alternative interpretations** is the richest among the four choices, as tracking more alternative interpretations increases the likelihood of identifying the overall most-likely interpretation and improves accuracy. However, exploiting this level of concurrency involves frequent synchronizations across numerous algorithmic steps. Researchers often exploit the easier source of concurrency in acoustic model computation before attempting to exploit this source of concurrency. Ravishankar (1993) first mapped this fine-grained concurrency onto the PLUS multiprocessor with distributed memory. You et al. (2009a) have proposed an implementation using OpenMP on a multicore system, with later work by You et al. (2009b) using task queues to map tasks to processors. Chong et al. (2009) and You et al. (2009b) have successfully exploited concurrency at this level on manycore accelerators, with thousands of concurrent contexts running at the same time.

Both sources of concurrency in the acoustic model computation and across alternative interpretations are scalable, enabling continued speedup as implementation platforms become more parallel. They can also improve recognition latency, which opens up new application areas in real-time recognition. Thus, we focus on exploiting these two sources of concurrency in this chapter. It is worth noting that the concurrency at the sentence level is orthogonal and can be applied across clusters of multicore and manycore computation nodes to achieve additional improvements in recognition throughput.

Looking back to Figure 1.2, in this section we have presented our approach to Concurrency Identification; we now continue the flow by examining the Software Architecture and Challenges in the following section.

1.3 Software Architecture and Implementation Challenges

Once the suitable sources of concurrency are identified, a software architecture can be designed to exploit them. We define a software architecture as a hierarchical composition of *patterns*. *Patterns* are solutions to recurring design problems that domain experts learn (Keutzer and Mattson (2009)). Figure 1.3 illustrates one such software architecture constructed to exploit both the concurrency in acoustic model computation and across alternative interpretations.

As shown in Figure 1.3, the inference engine implements the Viterbi search algorithm, which finds the most likely word sequence considered to be the "hidden cause" of the acoustic signal. At the top level, the algorithm employs the *Iterator pattern* - the computation iterates through a sequence of feature vectors extracted from the acoustic signals one time-step per iteration. Within each iteration, the

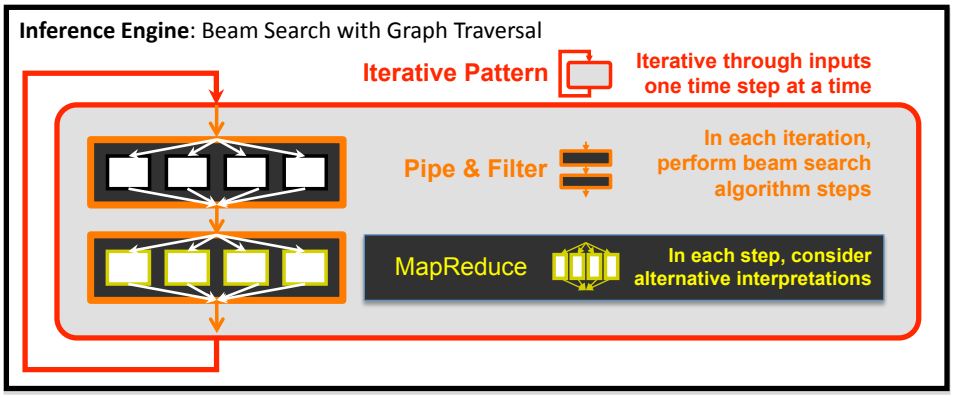

Figure 1.3 Software architecture of the automatic speech recognition inference engine

algorithm goes through two phases of execution sequentially in a *Pipe & Filter pattern*. The first phase is compute-intensive and estimates the observation probability of the feature vector with respect to a set of acoustic features. The features are represented as sets of Gaussian mixtures in the feature space. The second phase is communication intensive, during which the algorithm manages the likely alternative interpretations of the input feature sequence. In our weighted finite state transducer (WFST) based speech model, Phase 2 is a traversal through a graph with probabilistic state-to-state transitions. The sources of concurrency being exploited lie inside Phase 1 and Phase 2 in the iteration loop. We exploit this concurrency using the *MapReduce pattern*, where each state-to-state transition is mapped to an execution unit, and the results are accumulated (i.e. reduced) at the end of the computation.

This software architecture cleanly exploits the target concurrency sources, but its implementation has many challenges. We highlight five areas here:

1. **Frequent synchronizations**: Phase 1 and Phase 2 of the inference algorithm consist of multiple algorithmic steps that require global synchronization between them. In ASR for example, one step may be computing likelihood of Gaussian mixtures in the acoustic model while the next step is computing a set of arc traversals based on results from the previous step. This means that all Gaussian mixture computations must complete at a global barrier before any computation in the arc traversal step can begin. Such global synchronization represents an expensive process of data sharing among multiple cores. This synchronization, when occurring frequently, could dominate total execution time. For example, guided by Amdahl's Law, if an application has 25% of its exe-

cution time taken up by synchronization, the application cannot be sped up by more than 4×, even with infinite computation resources.

2. **Large data working set**: During every iteration of the inference algorithm, more than 100 MB of Gaussian mixture model parameters could be referenced in Phase 1. In Phase 2, the range of data that could potentially be accessed exceeds 400 MB. This working set size is beyond the scope of on-chip cache hierarchies. Any techniques to reduce the amount of data accessed, or improve the efficiency with which this data is accessed would be beneficial.

3. **Low computation to communication ratio**: Many of the algorithm steps involve collecting parameters from various models to infer the likelihood of an observation or manage an alternative interpretation. This process requires few floating point operations of computation and many data accesses. A low computation to communication ratio implies that the application is likely to be bottlenecked by the available communication bandwidth, rather than the computation throughput. On today's highly parallel platforms, the processing units often have significant processing power but have to share the channel to off-chip memory. Thus for such platforms a low computation to communication ratio indicates that it would be hard to fully exploit the capabilities of the processing unit.

4. **Irregular data structure**: The WFST speech model used in Phase 2 is an irregular graph that could contain millions of states and arcs. The number of out-degrees from states in the directed graph ranges from 1 to 897, and the number of in-degrees could be more than 16,000. The distribution of the in-degrees and out-degrees will vary with respect to different vocabulary and languages. This will make it difficult to optimize on platforms with wide vector units, where efficient execution depends on regularity of data accesses and computation.

5. **Unpredictable workload and memory access pattern**: The traversal through the WFST speech model represent alternative interpretations and is dependent on the acoustic input. This makes the workload size and memory access patterns highly unpredictable.

Having constructed a software architecture and identified challenges in the implementation, we now discuss how to map the software architecture to parallel platforms (following the process in Figure 1.2). In the next section, we demonstrate ASR implementations for a multicore and a manycore platform and address the programming concerns in efficiently exploiting the identified sources of concurrency.

1.4 Efficient Implementations on Parallel Platforms

An efficient implementation on a parallel platform should be aware of all the resources and limitations of the platform. Nuances in hardware architecture components such as memory hierarchy often require the use of different data structure alternatives. They could also require different implementations of tasks and different runtime mechanism to load balance the parallel tasks. In short, depending on the implementation platform, the parallelization approach could be very different.

For ASR, we discuss various data, task and runtime considerations in implementing efficient solutions on multicore and manycore platforms. Specifically, we use an Intel Core i7 multicore platform and an NVIDIA GTX280 manycore platform, the specifications of which are compared in Table 1.2. We consider multicore processors as processors that devote significant transistor resources to complex features for accelerating single thread performance. On the other hand, manycore processors use their transistor resources to maximize total instruction throughput at the expense of single thread performance.

Table 1.2 *Parameters for the experimental platforms*

Type	Multicore	Manycore
Processor	Core i7 920	GTX280 (+Core2 Q9550)
Cores	4 cores (SMT)	30 cores
SIMD Width	4 lanes	8 physical, 32 logical
Clock Speed	2.66 GHz	1.296 GHz
SP GFLOP/s *	85.1	933
Memory Capacity	6GB	1GB (8GB)
Memory BW	32.0 GB/s	141.7 GB/s
Compiler	icc 10.1.015	nvcc 2.2

* Single precision giga floating point operations per second (SP GFLOP/s)

The GTX280 has almost an order of magnitude more cores than the Core i7, but each core runs at half the clock frequency than that of the Core i7. Leveraging the wider SIMD unit, the theoretical peak single precision floating point operation throughput of the GTX280 is over an order of magnitude more than that of the Core i7. However, the microarchitecture of the GTX280 is more restrictive, limiting the achievable throughput to approximately three to six times the throughput of the Core i7. In terms of access to data in off-chip volatile memory, GTX280 has more than four times the bandwidth compared to the Core i7. However, it has a less flexible on-chip cache hierarchy that does not include hardware cache coherency

support between cores, which increases the bandwidth pressure on the off-chip memory bus.

These differences in the platform specifications heavily influence the data, task and runtime considerations in designing efficient implementations of ASR. In the following sections, we describe specific platform characteristics that are driving design decisions and illustrate how the application challenges are met and resolved. Where relevant, we also describe the alternative implementations we experimented with, and why they did not perform as well for ASR.

1.4.1 Multicore Infrastructure and Mapping

In this section, we discuss various data, task and runtime considerations in implementing an efficient solution on the Core i7 multicore platform and conclude with the overall flow chart of the implementation.

Data Considerations

The Core i7 multicore processor has four cores, each has a dedicated 64KB (32I + 32D) L1 and 256 KB L2 cache. There is also a shared unified 8MB L3 cache. The core-specific L1 and L2 caches call for data locality considerations, and frequent data transfers between cache levels make cache alignment important.

1. *Data Locality: Phase 1 Evaluation*

   GMM evaluation in Phase 1 requires significant number of memory accesses to load Gaussian parameters. Evaluation of the whole GMM would require 10-100 MB of Gaussian parameter data to be loaded at each iteration in Figure 1.3. Although the actual amount of Gaussian parameters to be utilized in Phase 1 can be reduced by the beam search strategy by as much as 35% in our speech model, the data size is still large enough to hinder the cache locality between consecutive iterations. The working set of GMM evaluation migrates slowly across iterations such that Gaussian parameters utilized in current iterations is likely to be accessed in the next iteration. However, the multicore caches are not large enough to maintain data from one iteration to another, leading to capacity misses. Moreover, this could also displace the working set of Phase 2 in every iteration, eliminating the possibilities of cross-iteration sharing.

   For better utilization of the cache, we can load the Gaussian parameters as non-cachable streaming data, and speculatively evaluate GMM for multiple future iterations. This way, we explicitly manage the temporal locality in Phase 1 across iterations with un-cached data and also allow the Phase 2 working set to

reside in the caches across iterations. Although larger number of future iterations to evaluate brings more data locality, it also increases unnecessary computations due to the migrating working set of GMM. Thus, we need to find the optimal number of future iterations by performing experiments.

2. *Data Alignment: Phase 2 Evaluation*

   Memory accesses are optimized for 64-byte alignment, which is the size of one cache line. Access to unaligned data is costly both from memory bandwidth utilization perspective and execution efficiency perspective, as unaligned load or store instruction are expanded to multiple micro-ops which reduce the throughput of this type of instruction in the processor execution pipeline.

   In ASR, the speech model is represented by a graph, where each state and each arc has properties associated with them. State and arc accesses are input dependent and difficult to pack or align. To avoid unaligned memory access penalties, we choose an array-of-structs data structure for the state. Each state is cache-line-aligned in memory and is stored along with the associated properties in structs that are exactly one cache line in size. For arcs, the information from a source state is stored consecutively in the main memory, in a structs-of-arrays layout. This layout reduces the memory access time of Phase 2, since the outgoing arcs from a source state are accessed consecutively in the same thread.

Task Considerations

1. *Task Granularity*

   The Core i7 multicore platform has 4-wide vector units that allows one vector instruction to simultaneously operate on four 32-bit data element. The vector operations are also called SIMD operation for "Single Instruction Multiple Data". SIMD efficiency is the ability of all vector lanes to synchronously execute useful instructions. When all lanes are fully utilized for an instruction, we consider the execution "synchronized" and the computation is load balanced at the SIMD level. When operations are not synchronized, we consider the execution "divergent" and the computation becomes unbalanced with some lanes sitting idle while others do useful work. In order to get the best performance of the platform, all vector lanes should execute in a "synchronized" fashion, thus we should try to assign the same amount of computation to each vector lane.

   Phase 1 in the inference process involves computationally intensive Gaussian Mixture Model (GMM) evaluation. For the evaluation of each Gaussian mixture, we can assign the mixtures to the lanes in the SIMD instruction. As long as the number of mixtures is a multiple of four, we achieve efficient utilization of Core i7's vector unit.

   Phase 2 in the inference process is more complex. As shown in Figure 1.4, a typical implementation is to evaluate each active state in the speech model on a

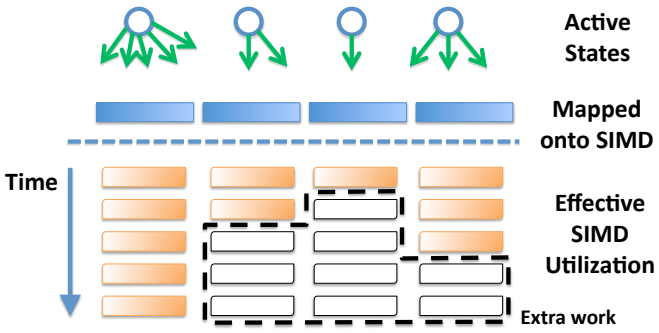

Figure 1.4 SIMD under-utilization in a state-based traversal

lane in the SIMD instruction. We call this approach *state-based* graph traversal. In this approach, the out-degrees of the states vary widely, which often results in "divergent" execution in the vector unit. On the other hand, there are overheads if we explicitly collect all the arcs to be evaluated and distribute each arc to a vector lane to evaluate in a "synchronized" fashion. We call this approach *arc-based* graph traversal. Our experiments show that on our speech model, the *state-based* and *arc-based* approaches achieved similar performances on Core i7, with the *state-based* approach being slightly faster. On architectures with wider SIMD instruction sets, however, "divergent" execution will incur higher penalties, and the *arc-based* approach is expected to be faster.

2. *Synchronization Cost*

   The Core i7 platform supports basic atomic operations by either adding "LOCK" prefix to integer instructions or directly using special instructions such as compare-and-swap (CMPXCHG) Intel (2009). When potential write conflicts exist in the multi-threaded algorithm, we can implement efficient synchronizations between cores using these instructions.

   During Phase 2 of the inference process, each core evaluates state-to-state transitions and updates destination states if necessary. The destination state is updated only when a transition provides higher probability than all prior transitions considered. There may be multiple cores trying to update the same destination state during the execution, which results in potential write conflicts. We utilized the compare-and-swap operation to synchronize destination state update. First, the current value of the destination state is fetched. Then, the evaluated transition probability is compared with the current value. Finally, if the transition probability is higher than the current value, the destination state value is updated by the transition probability with a compare-and-swap instruction.

   This *propagation-based* approach, where results are propagated from source

states to destination states with atomic operations, can be significantly more efficient compared to software-managed data-parallel write conflict resolution mechanisms. We also experimented with an *aggregation-based* software-managed approach, where unique result buffers are created for each state-to-state transition. The result buffers associated with the same destination state are explicitly reduced at the end of all transition evaluations to obtain the most likely state-to-state transition at each end state. On the Core i7 platform, the *propagation-based* approach was an order of magnitude faster than the *aggregation-based* approach for Phase 2 of the algorithm, as illustrated in Section 1.5.

Runtime Considerations

1. *Task Scheduling and Load Balancing*

   The Core i7 multicore platform provides a shared memory abstraction that enables a variety of parallel programing abstractions. These include the POSIX Threads (see Butenhof (1997)), Cilk (see Blumofe et al. (1995)), OpenMP (see Chandra et al. (2000)) or light-weight task queue implementations such as CARBON by Kumar et al. (2007).

   For this parallelization of the ASR application, we have chosen a concurrency source that is most scalable, with fine-grained units of work at each algorithm step that are as short as 10-100s of instructions. At this grainularity, it is crucial to choose a parallel programming abstraction that is light weight in task generation and execution.

   For our multicore implementation, we chose CARBON, a distributed task queue programming framework by Kumar et al. (2007) where a task is a function that executes in one thread and can be scheduled as a unit. The application creates an array of tasks for arc or state computation and the framework assigns sections of the task array to available processors. The framework then monitors for idle cores that have completed their tasks early and load balances the system during run-time.

   Although the working set of the speech model migrates every iteration depending on input audio features, the working set size, i.e. the number of active states/arcs in the speech model, is only 1 - 2% of total speech model. Moreover, the working set on average overlaps by 80% between consecutive iterations. Thus, instead of distributing the tasks evenly among the set of task queues, we can assign each task to the thread queue where it was processed in the previous time frame. In this method, the initial workload is inevitably imbalanced when the processors start execution. However, the lazy task stealing guarantees the eventual load balance. Since the tasks are likely to be processed in the same processor for many iterations, we could achieve approximately 20% speedup in Phase 2 with a notion of affinity between tasks and processors.

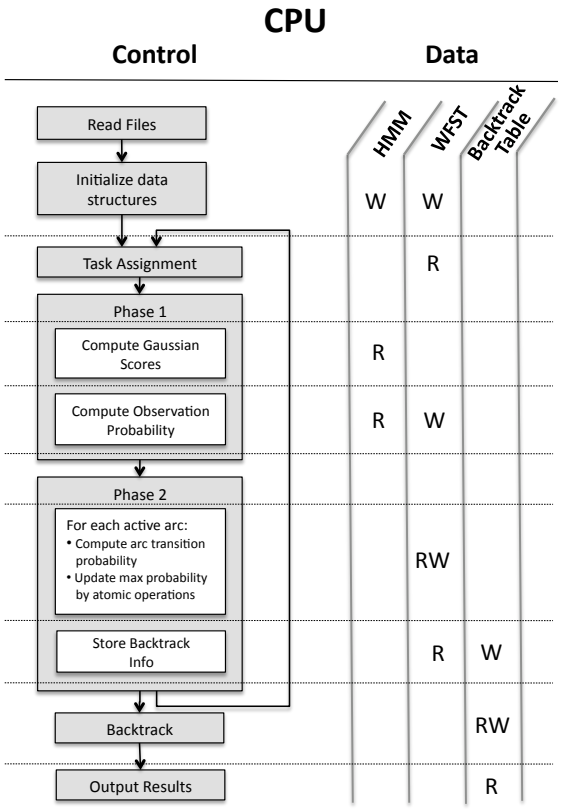

Figure 1.5 Summary of the data structure access and control flow of the inference engine on the multicore platform

To achieve high cache performance, we could also utilize a static scheduling method in which the speech model is partitioned offline and each processor executes on only its designated partition. However, it is difficult to statically partition the irregular graph while maintaining good load balancing.

Summary

Given the discussion on data, task and runtime considerations in implementing an efficient solution on the Core i7 platform, we present the final implementation flow in Figure 1.5.

In this implementation, all the data structures are stored in main memory and the working set is managed by the hardware cache hierarchy, which is a highly efficient low latency synchronization mechanism for the **frequent synchonizations**. To efficiently utilize the underlying cache architecture, we speculatively evaluate Gaussian mixtures for multiple future frames for increased temporal locality in Phase 1,

and mitigated issues with the **large data working set**. We also layout the data structure aligned with the cache lines to minimize data transfers between cache levels, and improved the **low computation to communication ratio**. Frequent synchronization during the graph traversal phase is implemented by efficient hardware-assisted atomic operations. Finally, we adopted task-queue-based dynamic task scheduling to deal with the variable task execution time caused by traversing an **irregular data structure**, and achieved good **workload balance** among multiple cores. Additionally, we enhanced the cache utilization and promoted temporal locality by establishing task to core affinity.

1.4.2 The Manycore Implementation

In this section, we discuss various data, task and runtime considerations in implementing an efficient solution on the GTX280 manycore platform and conclude with the overall flow chart of the implementation.

Data Considerations

1. *Memory Hierarchy*

   The GTX280 manycore platform has two levels of memory hierarchy for the GPU to access data with orders-of-magnitude differences in the memory throughput. The host-to-device memory accesses have 2.5GB/s data transfer rate and the device memory bandwidth is about 120GB/s. Graph traversal process of the inference engine has a highly irregular memory access pattern. Thus, it is essential to keep the working set in device memory for high bandwidth access. The GTX280 provides 1GB of device memory on the GPU board, which can fit the acoustic model (130MB), the language model (400MB) and various temporary graph traversal data structures. The GTX280 architecture has a less flexible memory hierarchy than the traditional cache-based architectures. The GTX280 has a global memory shared by all multiprocessors. Each multiprocessor also has a fast local shared memory space (16KB per multiprocessor) which is software-managed. In addition the memory hierarchy does not include hardware cache coherency support between cores, which increases the bandwidth pressure on the off-chip memory bus.

   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 CUDA kernels to utilize the high device memory bandwidth. 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 10MB of data per second, which translates to less than 5ms of transfer time on a channel with 2.5GB/s bandwidth and thus is a negligible fraction of the overall computation.

2. *Data Structure Regularity and Working Set*

   Data accesses on manycore platforms need to be extremely regular. Specifically, data accesses can be classified as "coalesced" or "uncoalesced". A "coalesced" memory access loads a consecutive vector of data that directly maps onto the vector lanes of the processing unit of the manycore platform. Such accesses efficiently utilize the available memory bandwidth. "Uncoalesced" assesses, on the other hand, load non-consecutive data elements to be processed by the vector units thereby wasting bandwidth. Thus, in order to fully utilize the manycore platform we must ensure memory accesses are coalesced by constructing our data structure accordingly.

   During the traversal process, we access an arbitrary subset of non-consecutive states or arcs in the speech model in each 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. Thus, we explicitly manage our working set to contain the current set of active states and arcs, ensuring coalesced memory accesses and data reuse.

Task Considerations

1. *Task Granularity*

   The GTX280 manycore platform has 8-wide physical 32-wide logical SIMD vector units. It is essential for an implementation to fully saturate the compute resources of these wide SIMD vector units to obtain good performance on the manycore platform.

   We use the *arc-based* approach as discussed in Section 1.4.1, where each SIMD vector lane evaluates one arc transition from state-to-state during Phase 2 of the algorithm. Each arc evaluation presents a constant amount of work, thus the evaluation process is "synchronized". This approach requires extra memory storage overhead as well as extra processing overhead to create more fine-grained tasks. For 32-wide SIMD operations, the performance benefit we get from "synchronized" execution including the processing overhead is more significant than the penalty incurred in "divergent" execution.

   The alternative approach to mapping execution tasks to SIMD vector units would be to assign each lane to evaluate a state (the *state-based* approach). This approach presents less software overhead. However, as shown in Figure **??**

for the state-based approach the control flow diverges resulting in an unbalanced computation as some lanes are idle, while other do useful work. For the GTX280 32-wide SIMD vector unit, the "divergent" control flow results in only 10% SIMD utilization. Using an *arc-based* implementation gives 9× speedup for the same computation, thus nearly gaining back full SIMD efficiency.

2. *Synchronization Cost*

   The GTX280 provides efficient atomic operations for between core synchronizations (CUD (2009)). Its atomic support goes beyond the standard Compare-AndSwap operations and includes some simple arithmetic and logic operations such as *atomicMax* . When used properly, these atomic operations can merge multiple high latency operations into a single atomic access, significantly improving application efficiency.

   During Phase 2 the task for each core is to evaluate state-to-state transitions. Multiple transitions can end in the same state. This creates a potential write-conflict in reading, comparing and saving the end state properties and eventually in maintaining the highest state-to-state transition probability. Using the *atomicMax* we solve all these issues by atomically updating the value of the end state only if the probability of the new transition is higher. The efficient atomic support on GTX280 reduces synchronization cost to a theoretical minimum: with only one operation per parallel task.

   The *propagation-based* approach described above propagates results from source states to destination states with atomic operations. It is significantly more efficient compared to software-managed data-parallel write conflict resolution mechanisms. We also experimented with a *aggregation-based* approach as discussed in Subsection 2, and observed a 2 millisecond overhead for using the *aggregation-based* write-conflict resolution compared with a 0.05ms overhead for using the *propagation-based* approach. Leveraging hardware-assisted atomic operations on the GTX280 resulted in an almost two orders of magnitude performance improvement.

Runtime Considerations

1. *Task Scheduling and Load Balancing*

   We use the CUDA programming framework (CUD (2009)) to implement the inference process. An application is organized into a sequential host program running on the host system(the CPU) and one or more parallel kernels running on the accelerators (the GPU). A kernel executes a set of scalar sequential programs across a set of parallel threads. The programmer can organize these threads into thread block, which are mapped onto the multiprocessing units on the GTX280 at run time. Task scheduling and load balancing are handled by

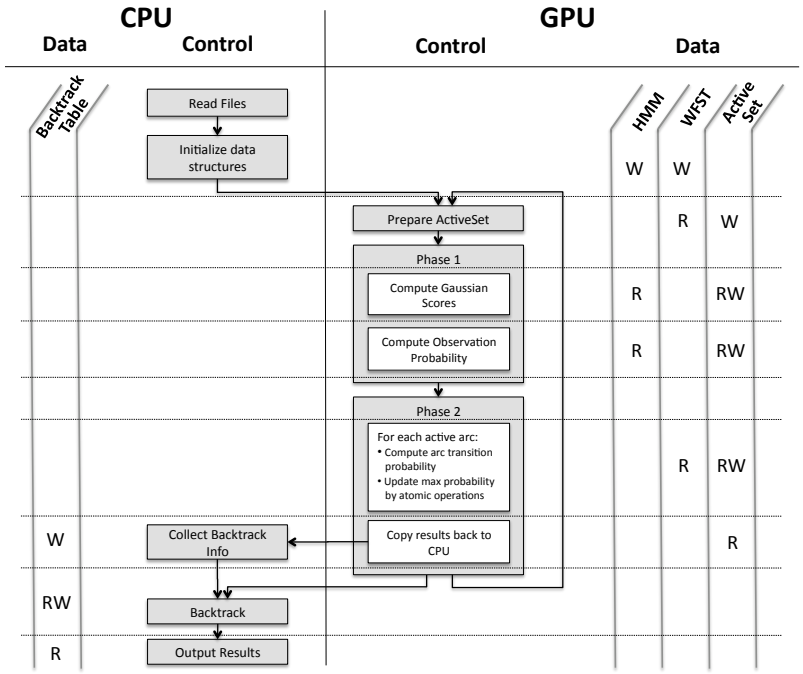

Figure 1.6 Summary of the data structure access and control flow of the inference engine on the manycore platform

the device driver automatically. In ASR, there is a significant amount of concurrency to allow for good load balance among the many cores at run time.

Summary

Given the discussions on data, task and runtime considerations in implementing an efficient solutions on the GTX280 manycore platforms, we present the final implementation on GTX280 in Figure 1.6.

In this implementation, we off-load the entire inference process to the GTX280 platform and take advantage the efficient hardware-assisted atomics operations to facilitate the challenge of having **frequent synchronizations**. The **large data working set** is stored on the 1GB dedicated memory on the GTX280 platform, and accessed through a memory bus with 140GB/s peak throughput. We start an iteration by preparing an ActiveSet data structure to gather the necessary operands into a "coalesced" data structure to maximize communication efficiency, and **improves computation to communication ratio**. We then use an *arc-based* traversal to handle the **irregular data structure** and maximize SIMD efficiency in the evaluation

of state-to-state transitions. Finally, we leverage the CUDA runtime to efficiently meet the challenge in scheduling the **unpredictable workload size** with variable runtimes onto the 30 parallel multiprocessors on GTX280.

Following Figure 1.2, after mapping the application to the parallel platform, we need to profile performance and do sensitivity analysis to different trade-offs particular to the specific implementation platform. We describe this process for our ASR inference engine application in the following section.

1.5 Implementation Profiling and Sensitivity Analysis

We have addressed the known performance challenges by examining data, task and runtime concerns and constructed a functionally correct implementation. Now we can analyze the performance achieved by these implementations.

1.5.1 Speech Models and Test Sets

Our ASR profiling uses speech models from the SRI CALO real time meeting recognition system (Tur et al. (2008)). The front end uses 13 dimensional perceptual linear prediction (PLP) features with 1st, 2nd, and 3rd order differences, is vocal-track-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 (Stolcke et al. (2008)). The acoustic model includes 52K triphone states which are clustered into 2,613 mixtures of 128 Gaussian components.

The pronunciation model contains 59K words with a total of 80K pronunciations. We use a small back-off bigram language model with 167k bigram transitions. The speech model 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 minutes in length and comprise 10 speakers. For the experiment we assumed that the feature extraction is performed offline so that the inference engine can directly access the feature files.

1.5.2 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

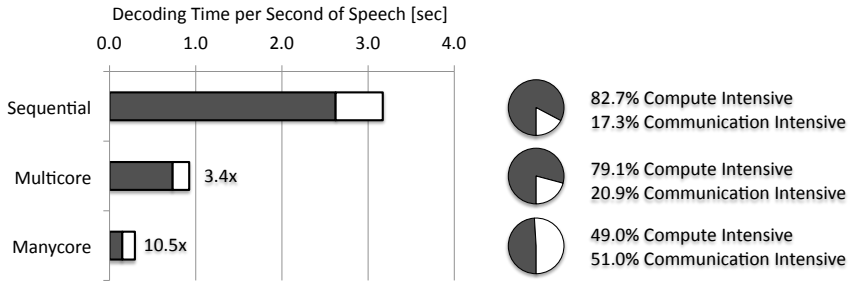

Figure 1.7 Ratio of computation intensive phase of the algorithm vs communication intensive phase of the algorithm

baseline is implemented on a single core in a Core i7 quadcore processor, utilizing a SIMD-optimized Phase 1 routine and non-SIMD graph traversal routine for Phase 2. Compared 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 Fig. 1.7 by highlighting the distinction between the compute intensive phase (black bar) and the communication intensive phase (white bar). The compute intensive phase achieves 3.6× speed up on the multicore processor and 17.7× on the manycore processor, while the communication intensive phase achieves only 2.8× speed up on the multicore processor and 3.7× on the manycore processor.

The speedup numbers indicate that the communication-intensive Phase 2 dominates the run time as more processors need to be coordinated. In terms of the ratio between the compute and communication intensive phases, the pie charts in Fig. 1.7 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 49.0% of the total runtime. The dominance of the communication intensive phase motivates further detailed examination of Phase 2 in our inference engine.

1.5.3 Sensitivity Analysis

In order to determine the sensitivity to different styles of the algorithm in the communication-intensive phase, we constructed a series of experiments for both multicore and the manycore platform. The trade-offs in both task granularity and core synchronization techniques are examined for both platforms. The design space for our experiments as well as the performance results are shown in Figures 1.8 and 1.9. The choice in task granularity has direct implications on load balance and task

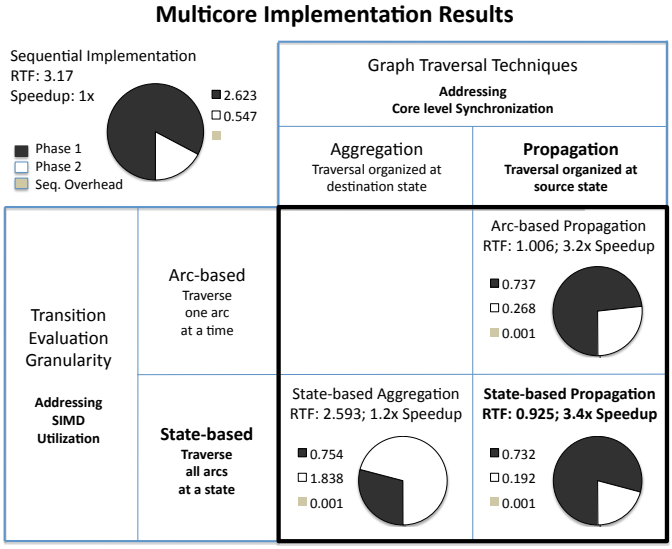

Figure 1.8 Recognition performance normalized for one second of speech for different algorithm styles on Intel Core i7

creation overhead, whereas the choice of the traversal technique determines the cost of the core level synchronization.

Each column in Figures 1.8 and 1.9 represents a different graph traversal technique and each row indicates different transition evaluation granularity. The Figures provide performance improvement information for Phases 1 and 2 as well as sequential overhead for all parallel implementation styles. The speedup numbers are reported over our fastest sequential version in the *state-based propagation* style. On both of the platforms the *propagation-based* style achieved better performance. However, the choice of best-performing task-granularity differed for the two platforms. For the manycore implementation, the load-balancing benefits of arc-based approach were much greater than the overhead of creating the finer-grained tasks. On the multicore architecture, the arc-based approach not only presented more overhead in creating finer-grained tasks, but also resulted in a larger working set thus increasing cache capacity misses. On wider SIMD units in future multicore platforms, however, we expect the *arc-based propagation style* will be faster than the *state-based propagation* style.

The Figures also illustrate that the sequential overhead in our implementation is less than 2.5% of the total run time even for the fastest implementations. This demonstrates that we have a scalable software architecture that promises greater

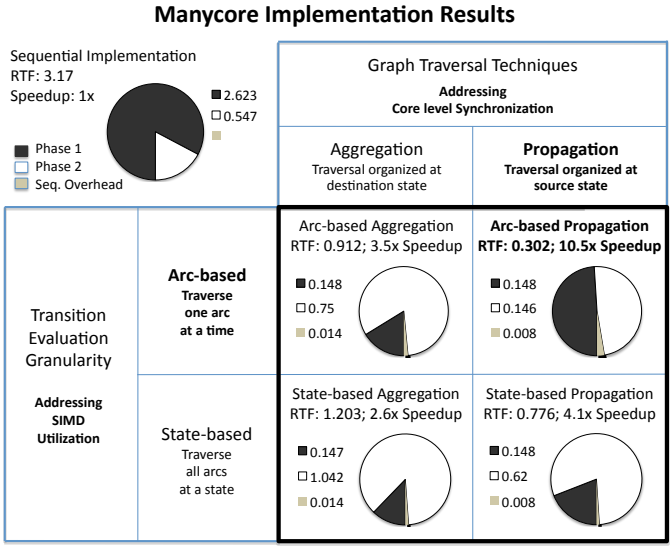

Figure 1.9 Recognition performance normalized for one second of speech for different algorithm styles on Nvidia GTX280

potential speedups with more platform parallelism expected in future generations of processors.

After performing the profiling and sensitivity analysis, we end up with a highly optimized implementation of the application on the parallel platform (see Figure 1.2). We can further optimize the implementation by making application-level decisions and trade-offs subject to the constraints and bottlenecks identified in the parallellization process. The following section describes an example of such optimizations.

1.6 Application-level Optimization

An efficient implementation is not the end of the parallelization process. For the inference engine on GTX280 for example, we observed that given the challenging algorithm requirements, the dominant kernel has shifted from the compute-intensive Phase 1 to the communication-intensive Phase 2 in the implementation. We have also observed that modifying the inference engine implementation style does not improve the implementation any further. In this situation, we should take an opportunity to re-examine possible application-level transformations to further mitigate parallelization bottlenecks.

1.6.1 Speech Model Alternatives

Phase 2 of the algorithm involves a graph traversal process through an irregular speech model. There are two types of arcs in a WFST-based speech model: arcs with an input label (non-epsilon arcs), and arcs without input labels (epsilon arcs). In order to compute the set of next states in a given time step, we must traverse both the non-epsilon and all the levels of epsilon arcs from the current set of active states. This multi-level traversal can impair performance significantly as each level requires multiple steps of cross-core synchronization. We explore a set of application transformation to modify the speech model to reduce the levels of traversal that is required, while maintaining the WFST invariant of accumulating the same weight (likelihood) for the same input after a traversal. To illustrate this, Fig. 1.10 shows a small section of a WFST-based speech model. Each time step starts with a set of currently active states, e.g. states (1) and (2) in Fig. 1.10, representing the alternative interpretations of the input utterances. It proceeds to evaluate all outgoing non-epsilon arcs to reach a set of destination states, e.g. states (3) and (4). The traversal then extends through epsilon arcs to reach more states, e.g. state (5) before the next time step.

The traversal from state (1) and (2) to (3), (4) and (5) can be seen as a process of active state wave-front expansion in a time step. The challenge for data parallel operations is that the expansion from (1) to (3) to (4) to (5) requires multiple levels of traversal. In this case, three level expansion is required, with one non-epsilon level and two epsilon levels. By flattening the epsilon arcs as shown in Figure 1.10, we arrive at the Two-Level WFST Model, where by doing one non-epsilon level expansion and one epsilon expansion, we can reach all anticipated states. If we flatten the graph further, we can eliminate all epsilon arcs and achieve the same results with one level of non-epsilon arc expansion.

Although model flattening can help eliminate the overhead of multiple levels of synchronization, it can also increase the total number of arcs traversed. Depending on the specific model topology of the speech model, we may achieve varying amount of improvements in the final application performance metrics.

1.6.2 Evaluation of Alternatives

We constructed all three variations of the speech model and measured both the number of arcs evaluated as well as the execution time of the communication-intensive Phase 2. We varied the amount of alternative interpretations, which is shown in Fig 1.11 as a percentage of total states that are active in the speech model.

The “L” shaped curves connect implementations that achieve the same recognition accuracy. At the application level, we are interested in reducing the execution

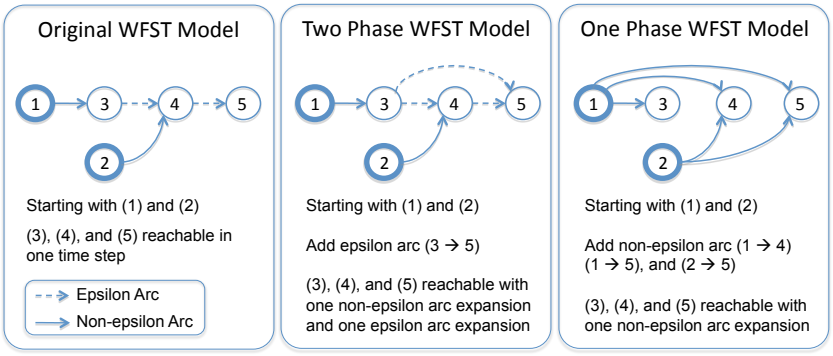

Figure 1.10 Model modification techniques for a data parallel inference engine

times for the communication intensive phase. Going from the Original setup to the Two-Level setup, we observe a large improvement in execution time, shown as a drop in the execution time graph of the communication-intensive phase. This execution time improvement was accompanied by a moderate increase in the number of arcs traversed during decoding, shown as a small shift to the right. Going from the Two-level setup to the One-level setup, we see a relatively smaller improvement in execution time, with a large increase in the number of arcs traversed.

An application domain expert who understands the implications of input formats on performance of the parallel application can make application-level transformation can further improve the application performance. For example in ASR, for the recognition task that maintains the smallest number of active arcs in this set of experiments, the speech model transformations are able to reduce the execution time of the communication intensive phase from 97ms to 75ms, and further to 53ms, thus almost doubling the performance for this phase.

1.7 Conclusion and Key Lessons

1.7.1 Process of parallelization

This chapter describes a process for the scalable parallelization of an inference engine in automatic speech recognition. Looking back at Figure 1.2, we start the parallelization process at the application level and consider the available concurrency sources in an application. The challenge is to identify the richest source of concurrency that improves performance given a particular application constraint such as latency or throughput (see Section 1.2). With the identified concurrency source, we construct the software architecture for the application using design patterns. Design patterns help create software architectures by composing structural

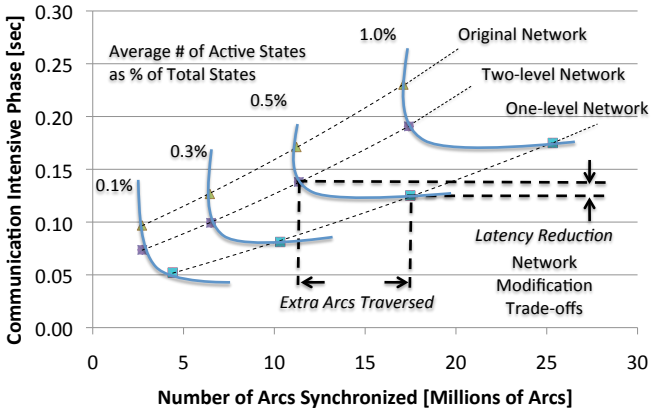

Figure 1.11 Communication Intensive Phase Run Time in the Inference Engine (normalized to one second of speech)

and computational patterns (Keutzer and Mattson (2009)) as shown in Section 1.3. The design patterns help identify the application challenges and bottlenecks in a software architecture to be addressed by the implementation.

The detailed implementation of the software architecture is performed with the consideration of three areas of concern (data, task and runtime) for a particular platform. The most effective parallel implementation strategy must recognize the implementation platform's architecture and leverage available infrastructure. Some of the areas of concern are taken care of by the infrastructure or the runtime system of the platform. In other cases, various styles of implementation strategy must be explicitly constructed as a series of experiments to determine the best decision for a particular trade-off. In that case a sensitivity analysis of the different implementation strategies must be performed.

Performance of an application can be improved by modifying the algorithm based on application domain knowledge. As illustrated in Section 1.5, the speech domain expert can make application-level decisions about the speech model structure while still preserving logical correctness. By identifying bottlenecks in the current implementation of the application, the domain expert can choose to modify the parameters of the application in order to make the application less sensitive to parallelization bottlenecks.

1.7.2 Enabling efficient parallel application development using frameworks

In order to develop a highly optimized implementation one needs to have strong expertise in all areas of the development stack. Strong application domain exper-

tise is required to identify available application concurrency as well as to propose application-level transformations that can mitigate software architecture challenges. Strong parallel programming expertise is required in developing a parallel implementation, where one needs to articulate the data, task and runtime considerations for a software architecture on an implementation platform. This complexity increases the risks in deploying large parallel software projects as the levels of expertise vary across the domains.

Our ongoing work on software design patterns and frameworks at the PALLAS group in the Department of Electrical Engineering and Computer Science at University of California, Berkeley attempts to address this problem by encapsulating the low-level parallel programming constructs into frameworks for domain experts. The PALLAS group believes that the key to the design of parallel programs is software architecture, and the key to efficient implementation of the software architecture is frameworks. Borrowed from civil architecture, the term *design pattern* refers to a solution to a recurring design problem that domain experts learn with time. A software architecture is a hierarchical composition of *architectural* software design patterns, which can be subsequently refined using *implementation* design patterns. The software architecture and its refinement, although useful, are entirely conceptual. To implement the software, we rely on frameworks.

We define a *pattern-oriented software framework* as an environment built on top of a software architecture in which customization is only allowed in harmony with the framework's architecture. For example, if the software architecture is based on the *Pipe & Filter* pattern, the customization involves only modifying pipes or filters. We see application domain experts being serviced by application frameworks. These application frameworks have two advantages: first, the application programmer works within a familiar environment using concepts drawn from the application domain. Second, the frameworks prevent expression of many notoriously hard problems of parallel programming such as nondeterminism, races, deadlock, and starvation.

Specifically for ASR, we have tested and demonstrated this pattern-oriented approach during the process of designing this implementation. Patterns served as a conceptual tool to aid in the architectural design and implementation of the application. Referring back to Figure 1.2, we can use patterns from the software architecture to define a pattern-oriented framework for a speech recognition inference engine application. The framework will be able to encapsulate many data, task, and runtime considerations as well as profiling capabilities, and will be able to be extended to related applications. While this framework is our ongoing research, we believe that these software design patterns and the pattern-oriented frameworks will empower ASR domain experts, as well as other machine learning experts, to quickly construct efficient parallel implementations of their applications.

References

2009 (March). *NVIDIA CUDA Programming Guide*. NVIDIA Corporation. Version 2.2 beta.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. 1995. Cilk: An Efficient Multithreaded Runtime System. In: *Proceedings of the Fifth ACM SIGPLAN*.

Butenhof, D. R. 1997. *Programming with POSIX Threads*. Addison-Wesley.

Cardinal, P., Dumouchel, P., Boulianne, G., and Comeau, M. 2008. GPU Accelerated Acoustic Likelihood Computations. In: *Proc. Interspeech*.

Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. 2000. *Parallel Programming in OpenMP*. Morgan Kaufmann.

Chong, J., Gonina, E., Yi, Y., and Keutzer, K. 2009 (September). A Fully Data Parallel WFST-based Large Vocabulary Continuous Speech Recognition on a Graphics Processing Unit. Page 11831186 of: *Proceeding of the 10th Annual Conference of the International Speech Communication Association (InterSpeech)*.

Chong, J., Friedland, G., Janin, A., Morgan, N., and Oei, C. 2010. Opportunities and Challenges of Parallelizing Speech Recognition. In: *Submitted to the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*.

Dixon, P. R., Oonishi, T., and Furui, S. 2009. Fast Acoustic Computations Using Graphics Processors. In: *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*.

Dixon, P.R., Caseiro, D.A., Oonishi, T., and Furui, S. 2007. The Titech large vocabulary WFST speech recognition system. *Automatic Speech Recognition & Understanding, 2007. IEEE Workshop on*, 443–448.

Intel. 2009. *Intel 64 and IA-32 Architectures Software Developer's Manuals*.

Ishikawa, S., Yamabana, K., Isotani, R., and Okumura, A. 2006. Parallel LVCSR Algorithm for Cellphone-oriented Multicore Processors. In: *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*.

Keutzer, K., and Mattson, T. 2009. Our Pattern Language (OPL).

Kumar, S., Hughes, C. J., and Nguyen, A. 2007. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In: *Proc. Intl. Symposium on Computer Architecture (ISCA)*.

Ravishankar, M. 1993. *Parallel Implementation of Fast Beam Search for Speaker-Independent Continuous Speech Recognition*. Tech. rept. Computer Science and Automation, Indian Institute of Science, Bangalore, India.

Stolcke, A., Anguera, X., Boakye, K., Cetin, O., Janin, A., Magimai-Doss, M., Wooters, C., and Zheng, J. 2008. The SRI-ICSI Spring 2007 Meeting and Lecture Recognition System. *Lecture Notes in Computer Science*, **4625**(2), 450–463.

Tur, G., Stolcke, A., Voss, L., Dowding, J., Favre, B., Fernandez, R., Frampton, M., Frandsen, M., Frederickson, C., Graciarena, M., Hakkani-Tr, D., Kintzing, D., Leveque, K., Mason, S., Niekrasz, J., Peters, S., Purver, M., Riedhammer, K., Shriberg, E., Tien, J., Vergyri, D., and Yang, F. 2008. The CALO Meeting Speech Recognition and Understanding System. Pages 69–72 of: *Proc. IEEE Spoken Language Technology Workshop*.

You, K., Lee, Y., and Sung, W. 2009a. OpenMP-based Parallel Implementation of a Continuous Speech Recognizer on a Multi-core System. In: *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*.

You, K., Chong, J., Yi, Y., Gonina, E., Hughes, C., Chen, Y.K., Sung, W., and Keutzer, K. 2009b (November). Parallel Scalability in Speech Recognition: Inference engine in large vocabulary continuous speech recognition. Pages 124–135 of: *IEEE Signal Processing Magazine*.