A Software and Hardware Environment for Developing AI Applications on Parallel Processors

R. Bisiani

Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
[412] 268-3072

Abstract

This paper describes and reports on the use of an environment, called **Agora**, that supports the construction of large, computationally expensive and loosely-structured systems, e.g. knowledge-based systems for speech and vision understanding. **Agora** can be customized to support the programming model that is more suitable for a given application. **Agora** has been designed explicitly to support multiple languages and highly parallel computations. Systems built with **Agora** can be executed on a number of general purpose and custom multiprocessor architectures.

1. Introduction

Our long-term goal is to develop a software environment that meets the need of application specialists to build and evaluate certain kinds of heterogeneous AI applications quickly and efficiently. To this effect we are developing a set of tools, methodologies and architectures called **Agora** (marketplace) that can be used to implement custom programming environments.

The kinds of systems for which **Agora** is useful have these characteristics:

- *they are heterogeneous* - no single programming model, language or machine architecture can be used;
- *they are in rapid evolution* - the algorithms change often while part of the system remains constant, e.g. research systems;
- *they are computationally expensive* - no single processor is enough to obtain the desired performance.

Speech and vision systems are typical of this kind of AI applications. In these systems, *knowledge-intensive* and conventional programming techniques must be integrated while observing real time constraints and preserving ease of programming.

State-of-the-art AI environments solve some but not all of the problems raised by the systems we are interested in. For example, these environments provide multiple programming models but fall short of supporting "non-AI" languages and multiprocessing. Some of these environments are also based on Lisp and are therefore more suitable (although not necessarily limited) to shared memory architectures.

For example, some programming environments provide abstractions tailored to the incremental design and implementation of large systems (e.g. LOOPS [14], STROBE [16]) but have little support for parallelism. Other environments support general purpose parallel processing (e.g. QLAMBDA [11], Multilisp [13], LINDA [7]) but do not tackle incremental design (Linda) or non-shared memory computer architectures (QLAMBDA, Multilisp). ABE [10] and AF [12] are the only environments we are aware of that have goals similar to **Agora**'s goals. ABE has, in fact, broader goals than **Agora** since it also supports *knowledge engineering*.

Agora supports heterogeneous systems by providing a *virtual machine* that is independent of any language, allows a number of different programming models and can be efficiently mapped into a number of different computer architectures. Rapid evolution is supported by providing similar *incremental programming* capabilities as Lisp environments. Programs that run on the parallel virtual machine can be added to the environment and share the same data with programs that were designed independently. This makes it possible to provide an unlimited set of *custom* environments that are tailored to the needs of a user, including environments in which parallel processing has been *hidden* from the end user. Finally, parallelism is strongly encouraged since systems are always specified as parallel computations even if they will be run on a single processor.

Agora is not an "environment in search of an application" but is "driven" by the requirement coming from the design and implementation of the CMU distributed speech recognition system [5]. During the past year, we designed and implemented an initial version of **Agora** and successfully used it to build two prototype speech-recognition systems. Our experience with this initial version of **Agora** convinced us that, when building parallel systems, the effort invested in obtaining a quality software environment pays off manyfold in productivity. **Agora** has reduced the time to assemble a complex parallel system and run it on a multiprocessor from more than a six man-months to about one man-month. The main reason for this lies in the fact that the details of communication and control have been taken care of by **Agora**. Application research, however, calls for still greater improvement. Significant progress in evaluating parallel task decompositions, in CMU's continuous speech project, for example, will ultimately require that a single person assemble and run a complete system within one day.

This paper is an introduction to some of the ideas underlying **Agora** and a description of the result of using **Agora** to build a large speech recognition system. The current structure of **Agora** is the outcome of the experience acquired with two designs and implementations carried out during 1985. One of these implementations is currently used for a prototype speech recognition system that runs on a network of Perqs and MICROVAXES. This implementation will be extended to support a shared memory multiprocessor, Sun's and IBM RT-PC's by the end of the second quarter of 1986.

2. Agora's Structure

Agora's structure can be explained by using a "layered" model, see Figure 2-1 starting from the bottom.

[1]Many individuals have contributed to the research presented is this paper, please refer to the **Acknowledgements** section for a list of each contribution. This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5167, and monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-0163. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government.

[2]Unix is a Trademark of AT&T

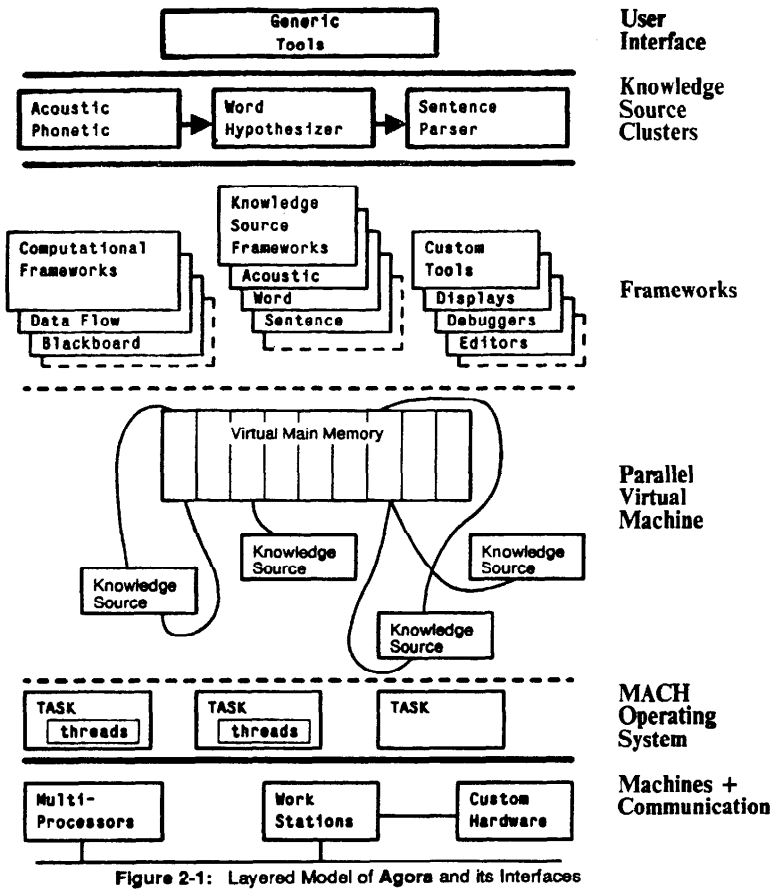

Figure 2-1: Layered Model of **Agora** and its Interfaces

-The first layer is a network of heterogeneous processors: single processors, shared memory multiprocessors, loosely-connected multiprocessors and custom hardware accelerators. The Mach operating system provides the basic software to execute computations on all these machines and **Agora** provides tools to map Mach abstractions into real machines. Mach is a Unix-compatible[2] operating system that runs on multiprocessors, see [2].

-The Mach layer provides three major abstractions: message passing, shared memory and threads. *Message passing* is the main communication mechanism: all **Agora** implementations can run on machines that provide message passing as the only communication mechanism. *Shared memory* (when available in the underlying computer system) is used to improve performance. *Threads* (processes that share the address space with other processes) are used to support the fast creation of new computations (a useful but not always vital characteristic).

-*The parallel virtual machine layer* represents the "assembly language level" of **Agora**. Computations are expressed as independent procedures that exchange data by using **Agora**'s primitives and are activated by means of a pattern matching mechanism. Computations can be programmed in either **C** or Common Lisp. It is in this layer that the most suitable Mach primitives are selected, the code is compiled and linked, tasks assigned to machines, etc. Computations expressed at this level are machine independent. Although systems can be fully described at this level, the virtual machine level is best used to describe *frameworks* rather than program user computations.

-*The framework layer* is the level at which most of the application researchers program. A framework is like a specialized environment built to interact with the user in familiar terms. The description, assembly, debugging and production run of an application system are all performed through its associated framework(s). Frameworks, as used in **Agora**, are very similar to ABE's frameworks, see [10].

First, *application engineers* program one or more "frameworks" that implement the programming environments that an application requires. A framework provides all the tools to generate and maintain a given kind of program. For example, a framework could contain virtual machine code to implement data-flow and remote-procedure-call communication mechanisms and a tool to merge user-supplied code with the existing virtual machine code. Such a framework could also contain a *structured* graphical editor that allows the user to deal with programs in terms of data-flow graphs.

Researchers can then use frameworks to create framework instantiations, i.e. frameworks that contain user provided code and data. Components of a framework instantiation can themselves be instantiations of some other framework. A framework instantiation can then be integrated with other framework instantiations to generate more complex frameworks. In the speech system, for example, the word hypothesizer is described by using a framework that embodies the asynchronous control necessary to run the word hypothesizer in parallel and code to display the data processed: a user need only be familiar with the algorithms and the language in which they are written in order to be able to experiment with different word hypothesization algorithms. A word hypothesizer generated in this way can be merged with an acoustic-phonetic framework instantiation by using, for example, a data-flow framework. Tools from all the original frameworks are made available in the combined framework.

We cannot describe frameworks in detail in this paper. Currently, we have implemented a "Unix-like" framework which lets users build parallel programs that communicate by using streams. We are implementing a data-flow framework that will let a user program through a graphic editor and a number of very specialized frameworks like the word hypothesizer framework described later.

3. The Agora Virtual Machine

The **Agora** virtual machine has been designed with two goals in mind: first, to be able to efficiently execute different programming models. Second, to avoid restricting the possible implementations to certain computer architectures. **Agora** is centered around representing data as sets of *elements* of the same type (elements can be regarded as variable-size records). Elements are stored in global structures called *cliques*. Each clique has a name that completely identifies it and a type (from a set of globally-defined types). **Agora** forces the user to split a computation into separate components, called Knowledge Sources (KS), that execute concurrently. Knowledge Sources exchange data through cliques and are activated when certain patterns of elements are generated. Any KS that knows the name of a clique can perform operations on it since **Agora** "registers" the name of each clique when it is created. Since "names" are global, the only requirement for sharing a clique between KSs is that a clique be first "created" by a KS and then declared "shared" by another KS. In a speech recognition system, for example, an element could be a phoneme, word, sentence or some other meaningful intermediate representation of speech; a clique could contain all the phonemes generated by a KS and a KS could be the function that scores phonetic hypotheses.

Element types are described *within the KS code* by using the syntax of the language that is used to program the KSs, with some additional information. The additional information is stripped from the source code by **Agora** before the code is handed to the compiler or interpreter. This means that users need not learn a different language. This is in contrast with other language-independent data transport mechanisms, like the mechanism described in [4], that use a separate language to define the data. The type declarations can contain extra information for scheduling and debugging purposes, e.g. the expected number of accesses per second, the legal values that elements can assume, display procedures, etc.

KSs can refer to sets of elements by using *capabilities*. Capabilities are manipulated by **Agora** functions and can be used to "copy" from a clique into the address space of a KS and viceversa (often no real copy will be necessary). There are two "modes" of access: *Read-only* and *Add-element*. Elements cannot be modified or deleted after they are written but Knowledge Sources can signal that they are not interested anymore in a given Element Clique.

Each KSs contains one or more user functions. KS functions are completely independent of the system they are used in and must only be able to deal with the types of element they use. KSs are created by calling an **Agora** primitive, each call to this function can generate multiple instances of the same KS. When a KS instance is created, a pattern can be specified: once the pattern is satisfied the KS function is activated. The pattern is expressed in terms of "arrival events" (the fact that an element has entered a clique) and in terms of the values of the data stored in the elements. For example, one can specify a pattern that is matched every time a new element enters the clique or that only matches if a field in the element has a specific value. More than one clique can be mentioned in the same pattern but no variables are permitted in the pattern (i.e. there is no binding). It is also possible to specify if an event must be considered "consumed" by a successful match or if it can be used by other patterns (this can be very useful to "demultiplex" the contents of a clique into different KSs or to guarantee mutual exclusion when needed).

A KS can contain any statement of the language that is being used and any of the **Agora** primitives, expressed in a way that is compatible with the language used. The Agora primitives are similar to the typical functions that an operating system would provide to start new processes (create new KS's in **Agora**), manipulate files (create, share and copy cliques) and schedule processes (i.e. change the amount of computation allocated to a KS).

KS's are mapped into Mach [2] primitives. In this mapping a KS can be clustered with other KS's that can benefit from sharing computer resources. For example, a cluster could contain KSs that access the same clique or KSs that should be scheduled (i.e. executed) together. Although clusters could also be implemented as Mach tasks, sharing the address space between "random" tasks can be very dangerous. Clusters can be implemented (in decreasing order of efficiency) as multiple processes that share memory or as multiple processes communicating by messages. Multiple instances of the *same* KS have a very effective and simple implementation on the Mach operating system [2] as a single process (task, in Mach terminology) in which multiple "threads" of computation implement the KSs.

Currently, the composition of clusters must be fully specified by the user, but **Agora** maintains information on which KSs are runnable and on how much of the cluster computation power each KS should be receiving. The computation power associated with a KS can be controlled by any KS in a cluster by using **Agora** primitives.

In conclusion, the **Agora** virtual machine provides mechanisms to statically and dynamically control multiprocessing: KSs can be clustered in different ways and executed on different processor configurations. Clusters can be used to dynamically control the allocation of processors. Therefore, **Agora** provides all the components necessary to implement focus-of-attention policies within a system, but the responsibility of designing the control procedures remains with the user.

4. Example of a System Built with Agora

We will illustrate how **Agora** can be used by describing the design of the CMU speech recognition system, ANGEL [5]. ANGEL uses more computation than a single processor could provide (more than 1,000 MIPS), is programmed in two languages (currently, the system comprises more than 100,000 lines of **C** and CommonLisp code), uses many different styles of computation, and is in continuous evolution since more than 15 researchers are working on it. Figure 4-1 shows the top level organization of the system. Arrows indicate transfer of both data and control. At the top level most of the components communicate using a data-flow paradigm, with the exception of a few modules that use a remote-procedure-call) paradigm. Eventually, a blackboard model will be used at the top

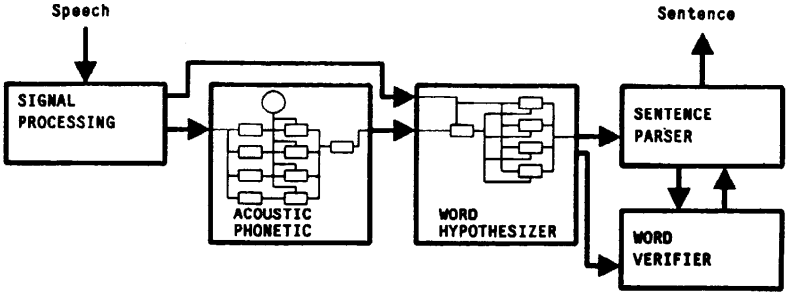

Figure 4-1: Structure of the ANGEL Speech Recognition System: Top-level and Internal Structure of the Acoustic-phonetic and Word-hypothesizer Modules

level. In Figure 4-1, two components contain a sketch of the structure of their subcomponents. For example, the acoustic phonetic component uses both data-flow and blackboard paradigms. It is important to note that Figure 4-1 shows the *current* structure and that part of the work being done on the system concerns expanding the number of modules and evaluating new ways of interconnecting them.

Frameworks are used to provide each component with the environment that is best suited to its development. At the top level there is a framework that provides a graphic editor to program data-flow and remote-procedure-call computations. Each subcomponent of the top level framework is developed using a different framework. We will use the word hypothesizer subcomponent as an example. The word hypothesizer generates word hypotheses by applying a beam search algorithm at selected times in the utterance. The inputs of the search are the phonetic hypotheses and the vocabulary. The times are indicated by a marker, called *anchor* that is computed elsewhere. The word hypothesizer must be able to receive anchors and phonemes in any order and perform a search around each anchor after having checked that all the acoustic-phonetic hypotheses within *delta* time units from the anchor are available. Phonetic hypotheses arrive at unpredictable times and in any order.

The word hypothesizer requires two functions: *the matching function* (**match()**) that hypothesizes words from phonemes and *the condition function* (**enough_phonemes()**) that checks if there are enough phonemes within a time interval from the anchor. The "editor" of the word-hypothesizer framework lets a researcher specify these two functions and binds them with the virtual machine level description. This description, that has been programmed by an application engineer, provides the parallel implementation. The framework also contains a display function that can be altered by a user.

The stylized code in Figure 4-2 describes the virtual machine level description used within the word hypothesizer framework. A speech researcher does not need to be aware of this description but only of the external specification of the two functions **match()** and **enough-phonemes()**. This description could be written in any language supported by **Agora** (currently **C** and Common Lisp).

```
Type declarations for cliques

KS setup
  Initialization: Create word and phoneme lattice clique
       Instantiate a few copies of KS word-hypothesize
       to be activated at the arrival of each new anchor
KS word-hypothesize
  Initialization: Declare the word, phoneme lattice and
       anchor cliques as shared
  Entry point: if enough_phonemes() then execute
       match() else instantiate KS wait to be
       activated at  each new phoneme
KS wait
  Initialization: Declare the word, phoneme lattice and
       anchor cliques as shared
  Entry point: if there are enough phonemes then match()
```

Figure 4-2: The Virtual Machine Level Implementation of the Word Hypothesizer

There are three Knowledge Sources: KS **setup** creates instantiations of KS **word-hypothesize** that are activated when "anchors" arrive. When any of these KSs receives an "anchor", it checks if there are enough phonetic hypotheses and, if so, executes **match()**. If not enough hypotheses are available, it creates an instantiation of KS **wait** that waits for all the necessary phonemes before executing **match()**.

The KS *word-hypothesize* can be compiled into a task if the target machine does not have shared memory. A parameter of the KS creation procedure indicates to **Agora** how many copies of the KS the framework designer believes can be efficiently used. If the machine has shared memory, then threads can be used and the parameter becomes irrelevant since new threads can be generated without incurring in too much cost. **Agora** can also be instructed to generate the **wait** KSs as threads of the same task. This is possible if KS **wait** and the functions it calls do not use any global data.

5. Custom Hardware for the Agora Virtual Machine

In a parallel system, the duration of an atomic computation (granularity) must be substantially bigger than the overhead required

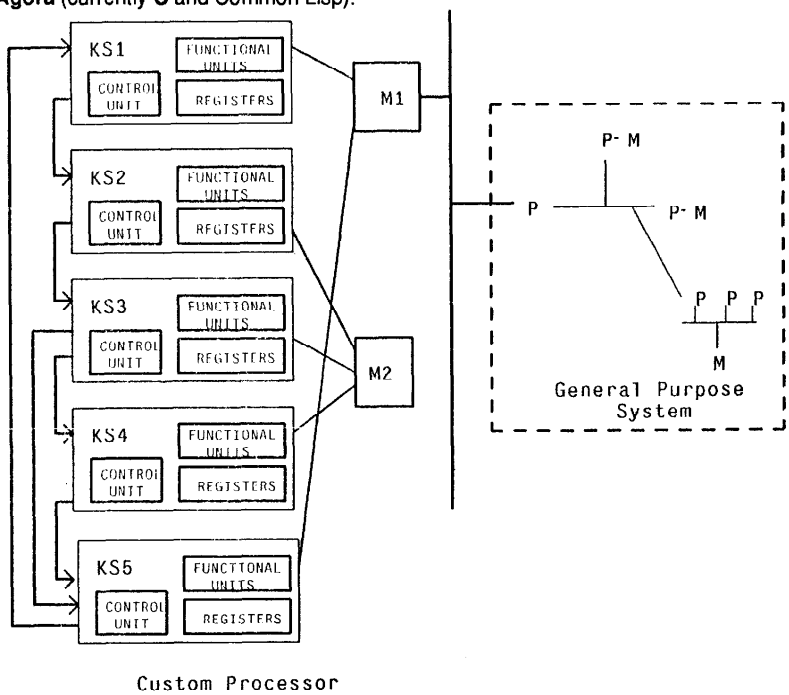

Figure 5-1: A custom hardware architecture for search

to start and terminate it. This overhead can be very large when using (conventional) general purpose architectures. **Agora** shares this characteristic with many other languages and environments. For example, Multilisp [13] cannot be used to efficiently implement computations at a very small granularity level unless some special hardware is provided to speed-up the implementation of futures. The effect of the overhead can be seen in the performance curves for the *quicksort* program presented in [13], Figure 5: the parallel version of the algorithm requires three times more processing power than the sequential version in order to run as fast (although it can run faster if more processing power is available). Therefore, hardware support *tailored to the style of parallelism provided by a language or an environment* is necessary. A proposal for an architecture that supports small granularity in concurrent Smalltalk can be found in [9].

The **Agora** virtual machine efficiently supports computations with a granularity larger than 500 ms when implemented on general purpose machines connected by a local area network. In the case of shared memory architectures, the limiting factor is the operating system overhead. Most of the current parallel environments, such as the Cosmic Cube [15] and Butterfly [3] environments, provide only minimal operating system functionality: typically, only the basic functions required to use the hardware. In these systems, granularity could be as small as a few hundred microseconds. **Agora** supports systems that are too large and complex to be implemented without a full operating system environment and must pay the price of operating system overhead. Therefore, the minimum granularity of an **Agora**'s KS on shared memory systems and Mach is about 10ms.

There is no single set of hardware which can lower the KS granularity independently of the computation being performed since each different style of computation poses different requirements on the virtual machine primitives. Therefore, our strategy is to develop architectures tailored to specific computations. So far, we have designed an architecture that supports computations that can be pipelined but are data-dependent and cannot be easily vectorized. For example, pipelines can be generated every time an algorithm executes a large number of iterations of the same loop. The architecture exploits the fact that data-flow control can be implemented very efficiently by direct connections between processors and the fact that a hardware semaphore associated with each element of a clique can speed-up concurrent access to elements. The architecture is *custom* because a set of KS's has to be explicitly decomposed and the architecture *configured* for that particular decomposition. This process is convenient when an algorithm is reasonably stable.

Figure 5-1 shows the structure of the architecture and how it interfaces with the rest of the system. Each KS is executed in a separate processor. Processors communicate through shared memory as well as through dedicated links. Dedicated links take care of data-flow while shared memories serve two functions: input/output and synchronized access to shared data to resolve data-

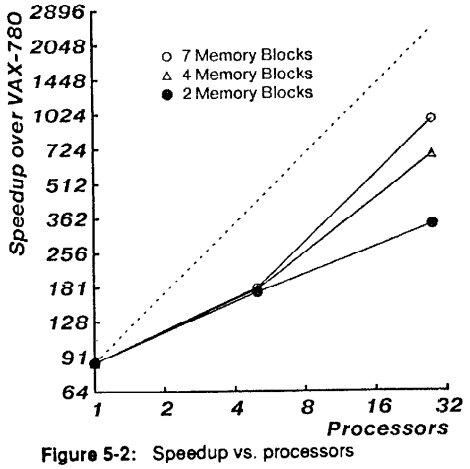

Figure 5-2: Speedup vs. processors

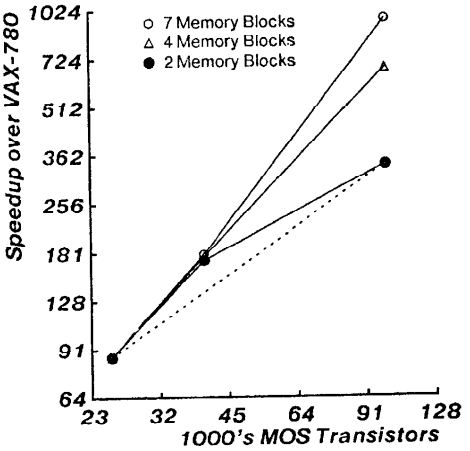

Figure 5-3: Speedup vs. transistors

dependencies.

Each processor contains: a simple hardwired control unit that executes a fixed program, functional units as required by the KS function, and register storage. An instruction can specify which data to read from local storage, what operations to perform on them, and where to store the result. A typical instruction can perform an addition or multiplication or access the shared memory. The architecture can be implemented as a fully-dedicated VLSI device or device set that communicates with the rest of the system through shared memory. Therefore, each processor can have different functional units and be wired directly to the processor(s) that follow it. A more expensive implementation with off-the-shelf components is also possible.

We evaluated the architecture by decomposing the *match* procedure used in the word hypothesizer described in the previous section. The match function uses a "best match, beam search" algorithm though other, more sophisticated search algorithms are already being planned. The current algorithm requires about 20 to 40 million instructions per second of speech with a 200-word vocabulary when executed in C on a VAX-11/780, depending on how much knowledge can be applied to constrain the search. A 5000-word vocabulary will require 500 to 1000 million instructions per second of speech.

We have simulated the custom architecture instruction-by-instruction while it executes the beam search algorithm with real data. The simulation assumed performance figures typical of a CMOS VLSI design that has not been heavily optimized (and therefore might be generated semiautomatically using standard cell design techniques). See [1] for details. The presence of 1, 2, 4, or 7 physical memory blocks was simulated to evaluate memory access bottlenecks. Figure 5-2 shows our simulation results as speedup relative to the performance of a VAX-11/780.

With five KS's and one physical memory we see a speedup of 170. This configuration could be implemented on a single custom chip by using a conservative fabrication technology (memories would still be implemented with off-the-shelf RAM's). With 28 KS's and seven memories, we can obtain speedups of three orders of magnitude, enough to cope with a 5,000 word vocabulary in real time. Moreover, each of the 28 processors is much smaller (in terms of hardware) than the original single processor and all 28 processors could share the same VLSI device by using the best available fabrication technology. This fact is illustrated graphically in Figure 5-3 which plots the speed-up against the transistor count of the design. The transistor count was obtained by adding the number of transistors in actual layouts of the various functional units and

registers, and is a crude estimate of the amount of silicon area required in a VLSI design.

6. Conclusions

Agora has a number of characteristics that make it particularly suitable for the development of complex systems in a multiprocessor environment. These include:

-the complexity of parallel processing can be hidden by building "reusable" custom environments that guide a user in describing, debugging and running an application without getting involved in parallel programming;
-computations can be expressed in different languages;
-the structure of a system can be modified while the system is running;
-KSs are activated by patterns computed on the data generated by other KS's;
-KSs are described in a way that allows **Agora** to match the available architecture and its resources with the requirements of the computation;
-custom architectures can easily be integrated with components running on general purpose systems.

Acknowledgements

The parallel virtual machine has been designed with Alessandro Forin [6, 8]. Fil Alleva, Rick Lerner and Mike Bauer have participated in the design and implementation of Agora. The custom architecture has been designed with Thomas Anantharaman and is described in detail in [1]. The project has also benefited from the constructive criticism and support of Raj Reddy, Duane Adams and Renato De Mori.

References

1. Ananthamaran, T. and Bisiani,R. Custom Search Accelerators for Speech Recognition. Proceedings of the 13th International Symposium on Computer Architecture, IEEE, June, 1986.

2. Baron, R., Rashid, R., Siegel, E., Tevanian, A., and Young, M. "Mach-1: An Operating System Environment for Large Scale Multiprocessor Applications". *IEEE Software Special Issue* (July 1985).

3. anon. *The Uniform System Approach to Programming the Butterfly(TM) Parallel Processor.* BBN Laboratories Inc., November 1985.

4. Birrel, A.D. and Nelson,B.J. "Implementing Remote Procedure Calls". *Trans. Computer Systems 2*, 1 (February 1984), 39-59.

5. Adams, D.A., Bisiani, R. The CMU Distributed Speech Recognition System. Eleventh DARPA Strategic Systems Symposium, Naval Postgraduate School, Monterey, CA, October, 1985.

6. Bisiani, R. et. al. Building Parallel Speech Recognition Systems with the Agora Environment. DARPA Strategic Computing Speech Workshop, Palo Alto, CA, February, 1986.

7. Carriero,N. and Glelernter,D. The S/Net's Linda Kernel. Proceedings of the Tenth ACM Symposium on Operating Systems Principles, December, 1985.

8. Bisiani,R. et.al. Agora, An Environment for Building Problem Solvers on Distributed Computer Systems. Proceedings of the 1985 Distributed Artificial Intelligence Workshop, Sea Ranch, California.

9. Dally, W.J. *A VLSI Architecture for Concurrent Data Structures.* Ph.D. Th., California Institute of Technology, March 1986.

10. Erman,L. et.al. ABE: Architectural Overview. Proceedings of the 1985 Distributed Artificial Intelligence Workshop, Sea Ranch, California.

11. Gabriel,R.P. and McCarthy,J. Queue-based multiprocessing Lisp. Symp. Lisp and Functional Programming, August, 1984.

12. Green,P.E. AF: A Framework for Real-time Distributed Cooperative Problem Solving. Proceedings of the 1985 Distributed Artificial Intelligence Workshop, Sea Ranch, California.

13. Halstead, H. "Multilisp: A Language for Concurrent Symbolic Computation". *ACM Trans. on Programming Languages and Systems 7*, 4 (Octber 1985), 501-538.

14. Bobrow,D.G. and Stefik,M.J. A Virtual Machine for Experiments in Knowledge Representation. Xerox Palo Alto Research Center, April, 1982.

15. Seitz, C. L. "The Cosmic Cube". *Comm. ACM 28*, 1 (January 1985), 22-33.

16. Smith,R.G. Strobe: Support for Structured Object Knowledge Representation. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August, 1983.