(4)

BBN Systems and Technologies Corporation

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 7034 AD-A208 196

Final Report
Contract No. N00039-85-C-0313
30 April 1985 - 28 February 1989

TOWARD REAL-TIME CONTINUOUS SPEECH RECOGNITION

Richard Schwartz and Owen Kimball

March 1989

Prepared for:

DARPA/ISTO
1400 Wilson Blvd.
Arlington, VA 22209

SPAWAR
Code 613
Washington, D.C. 20306

89 4 26 100

Report No. 7034

Final Report
Contract No. N00039-85-C-0313
30 April 1985 - 28 February 1989

TOWARD REAL-TIME CONTINUOUS SPEECH RECOGNITION

Richard Schwartz and Owen Kimball

March 1989

Prepared for:

DARPA/ISTO
1400 Wilson Blvd.
Arlington, VA 22209

SPAWAR
Code 613
Washington, D.C. 20306

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Final Report No. 7034	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Toward Real-Time Continuous Speech Recognition		5. TYPE OF REPORT & PERIOD COVERED Final Report 30 April 85-28 February 89
		6. PERFORMING ORG. REPORT NUMBER 7034
7. AUTHOR(s) Richard Schwartz and Owen Kimball		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS BBN Systems and Technologies Corporation 10 Moulton Street Cambridge, MA. 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS SPAWAR Code 613 Washington, D.C. 20306		12. REPORT DATE March 1989
		13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution of the document is unlimited. It may be released to the Clearinghouse, Dept. of Commerce, for sale to the general public.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

real-time speech recognition, parallel processing, Butterfly parallel processor.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We report on work toward real-time speech recognition. We investigated the use of the Butterfly parallel processor for increasing the speed of speech recognition. We found that it was possible to achieve factors of 10 speedup with minimal effort. More effort was required to achieve high efficiency on larger machines with 64 or more processors. We implemented a near-real-time demonstration of continuous speech recognition with a grammar.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Table of Contents

List of Figures

1. Executive Summary

This final report describes ~~our~~ research on real-time speech recognition. ~~We~~ have developed, under other DARPA-funded contracts, a system for continuous speech recognition. BYBLOS, the BBN Continuous Speech Recognition System, consists of a general paradigm and several algorithms for high performance speech recognition. The goals of the real-time project have been:

1. Develop algorithms and hardware for near-real time speech recognition,
2. Develop and assess techniques for using highly parallel machines for continuous speech recognition.

The focus of our work has been on parallel processing techniques for speech recognition. As such, most of our work has involved implementing different speech recognition algorithms on the Butterfly^{TM} Parallel Processor ("Butterfly"). Several demonstrations of different aspects of speech recognition were given using the Butterfly implementation. In addition, during the past year, we have devoted most of our effort under this contract supporting the joint project between SRI, UC Berkeley, and BBN to develop a set of special purpose speech recognition boards for real-time continuous speech recognition.

In the first year of the project (1985), we implemented an initial version of continuous speech recognition on a 16-processor Butterfly Parallel Processor. We used the Uniform System programming paradigm to allocate memory and computation among the different processors. After some revisions, the system was found to obtain 75% efficiency on 16 processors, which amounts to a speed up of a factor of 12 over the speed of a single processor.

In the second year we concentrated on improving the efficiency when a much larger number of processors was used. We used a Butterfly Parallel Processor with 97 processors. When the same code was used as on 16 processors, we found a speed up of a factor of 20, which was fast, but represented only 20% efficiency. We identified several issues of scheduling and memory contention that had not been significant on a smaller machine, but which began to dominate as we used larger configurations. After these issues were solved, we achieved a 79% efficiency, which amounts to an increase in speed by a factor of 77.

The work described above involved continuous speech recognition with no grammatical constraints. In the third year we implemented several different types of grammar-directed recognition algorithms on the parallel machine. These included bigram statistical grammars, deterministic finite-state automata (DFA) which are basic network grammars, nondeterministic finite-state automata (NFA), which are similar, but allow "null" arcs in the grammar, and the Word-Pair grammar, which is a special case of an NFA with a large number of null arcs. The special problems associated with using medium and larger grammars to constrain the sequence of words were solved. In addition, during the third year of the project, we implemented and gave several demonstrations of speech recognition using different types grammars.

Having demonstrated the feasibility of speech recognition using a highly parallel machine, we turned our attention to providing real-time speech recognition in a smaller, possibly portable device. After considering several commercially available boards, we decided to cooperate with SRI and UC Berkeley to help them build a set of special-purpose boards that would support speech recognition in real time. We consulted with them concerning the details of our recognition algorithm, and together with them determined an overall architecture that was both feasible and would accomplish the recognition task efficiently. We also helped by designing a small number of the VLSI cells for the special-purpose chips. Finally, we specified the hardware requirements for the front end signal processing board and wrote and tested all of the signal processing software for the TMS320C25 in assembly language.

Details of the work are given in the body of the report that follows.

2. Speech Recognition on the Butterfly Parallel Processor

We have developed, under other DARPA-funded contracts, a system for continuous speech recognition. BYBLOS, the BBN Continuous Speech Recognition System, consists of a general paradigm and several algorithms for high performance speech recognition. Our primary focus during this contract has been to demonstrate the effectiveness of using large-scale parallelism for near real-time continuous speech recognition. We started with the BYBLOS system for continuous speech recognition, which was being developed under other DARPA-funded contracts. The algorithm had inherent parallelism which we felt could be exploited on a parallel processor. This work was performed using the Butterfly^{TM} Parallel Processor (which will be referred to as "Butterfly" during the remainder of this text) because it was one of the few readily available and extensible general purpose parallel processors at the time. In this chapter we describe each of the algorithms implemented on the Butterfly.

In Section 1 we give a brief description of the Butterfly hardware and the Uniform System paradigm that we use to program it. Our first attempt at using a 16-processor configuration for continuous speech recognition with no grammar is outlined in Section 2. When we increased the size of the configuration to 97 processors we found (as expected) that the speed increased only a small amount. We were able to determine the causes of inefficiency in our use of large-scale parallelism, and by making several small modifications we restored the efficiency. These experiments are described in Section 3. In Section 4 we explain the special problems associated with using each of several different types of grammars to constrain the sequence of words recognized.

2.1 The Butterfly Parallel Processor

Hardware

The Butterfly Parallel Processor [1] is composed of multiple (up to 256) identical nodes interconnected by a high-performance switch. Each node contains a processor and memory. The switch allows each processor to access the memory on all other nodes. Collectively, these memories form the shared memory of the machine, a single address space accessible to every processor. All interprocessor communication is performed using shared memory. Typically memory referencing instructions accessing remote memory take about three times as long as references to local memory.

The Butterfly Parallel Processor is a multiple instruction multiple data stream (MIMD) machine in which each processor node executes its own sequence of instructions, referencing data as specified by the instructions. Processor Nodes are tightly coupled by the Butterfly switch. Tight coupling permits efficient interprocessor communication and allows each processor to access all system memory efficiently. The Butterfly Parallel Processor is expandable to 256 Processor Nodes. Each Processor Node contains a Motorola MC68000 family microprocessor, an optional floating-point coprocessor, from 1 to 4 MBytes of main memory, a co-processor called the Processor Node Controller, memory management hardware, an I-O bus, and an interface to the Butterfly switch. The particular machine that was used for development in this project was a 16 processor machine with 1 Mbyte of memory on each Processor Node. It did not have hardware support for floating point arithmetic.

Uniform System

We used the Uniform System approach to obtain a parallel implementation. The Uniform System is a programming methodology supported by a library of high-level functions [3]. It exploits the uniform environment provided by the architecture of the Butterfly Parallel Processor to simplify the problem of load balancing for the memory as well as for the processors. Memory accesses must be organized to avoid memory contention. The load on the processors is balanced when all processors are equally busy and no processor is waiting for another to finish.

Balancing the load on memory is accomplished by spreading out the data evenly across the different physical memories in the machine, under the assumption that this will also spread the accesses fairly evenly, reducing the inefficiency that results when many processors attempt to access the same memory simultaneously. Functions for allocating storage in the shared memory are included in the Uniform System, as are functions that perform block transfers between shared memory and local memory.

The philosophy behind the Uniform System processor management methodology views the processors as a uniform pool of workers, all of which know how to execute the same tasks. Using this methodology, the programmer is only required to supply code that operates correctly when multiple processors execute it at the same time. The processor management is accomplished by first copying the program to each of the processors. In most cases, the program will begin with a section of serial code that is executed on a single processor. To begin executing a section of code on multiple processors -- a FOR loop, for example -- the programmer can use a "task generator" to replace the FOR statement and a "worker routine" to replace the body of the FOR loop. The task generator makes a task descriptor available to all processors, which use it, as they become free, to generate calls to the worker routine. Processors, using this descriptor, execute the routine repeatedly for different index values, until the index has run its range. When all processors have finished, the program, once again serial, continues executing on a single processor.

We decided to use the Uniform System for several reasons. First, the speech recognition algorithm is essentially a single task, executed many times. This fits the Uniform System paradigm very well. Second, being novices, we were attracted by the simplicity of use of the Uniform System. Third, functions in the Uniform System allow automatic timing of the same program run on various numbers of processors, and this provided an easy way of evaluating the performance of the parallel implementation. Finally, because the same program can be run on one or many processors, we believed that debugging the parallel implementation would be simplified.

2.2 Initial Parallel Implementation

In this section we review the algorithm for continuous speech recognition. Then we explain how this algorithm can be mapped onto the Butterfly parallel processor using the Uniform System.

2.2.1 Speech Recognition Algorithm

To recognize speech we model it using a statistical approach. We treat the sequence of short term spectra as if they were the output of a hidden Markov model (HMM). The forward-backward algorithm can be used to estimate the parameters ofthe models that best describe a set of training data. Then, given an unknown sentence we use the Viterbi algorithm to determine the sequence of states of the HMM most likely to have produced the sequence of short term spectra in the unknown utterance.

The basic recognition algorithm finds the path through the states that is most likely to have produced the spectral sequence to be recognized. The algorithm does this by finding the best path to every state at every time given that the path to the previous state was also "best". Each step along these paths has associated with it a "score", which reflects the probability of the step given the spectral sequence and the transition probabilities of the model. The scores are accumulated along the paths, so that, at every time, the best path to any state has a single score. The best path to a particular state, S, at time t, is determined by considering all possible predecessor states (states which have transitions to S) at time t-1 and the best path to each of these. The score for a path to S is then the combination of the path score to the best predecessor state and the score for the step from that predecessor state to S.

The central computation in the algorithm is: for each time interval, update the scores for all states. Figure 1 schematically illustrates the scoring procedure for a single state in a word. In this figure, the score for state n at time t (the $\alpha_n(t)$ in the lower right corner) is being computed based on the scores for three states computed at time t-1 (the three circles on the left of the figure) each multiplied by the corresponding transition probability of going to state n. The new score for state n is just the maximum of the entering scores multiplied by the probability of the input spectrum at time t, given that the state is state n.

This scoring procedure is applied to all states in each word for all time frames in the utterance. The scores computed for terminal states (ends of words) are special. The score and start time of the word with maximum terminal score are saved. The largest terminal score for a time frame is used as the score for all word-initial states in the next time frame. In addition, the current state score,$\alpha_n(t)$ is compared against the largest state score, $\alpha_{Best}(t)$, encountered so far for the current time frame, and replaces it, if appropriate. This is used to derive a normalization factor ($NF = 1/\alpha_{Best}(t)$), which is used to prevent arithmetic underflow.

When all the time frames in the utterance have been processed in this way, the best sequence of words is determined. The maximum terminal score at the end of the utterance specifies the last word of the utterance. The start time of this last word is the end time for the previous word, so we can determine the second-to-last word in the theory, and so on, back to the beginning of the utterance.

2.2.2 Parallel Implementation

High speed computations require that no disk is used. Therefore all the necessary memory of the models of the words is distributed among the memory of all the processors.

The basic parallel task was defined to be the updating of the scores for all the states of one word. The pertinent portion of the speech recognition program can be abstracted as follows:

```
a) FOR all frames
b)   initialize frame
c)   FOR all words
d)     initialize word
e)     FOR all states
f)       compute state score
g)       IF (new max score)
```

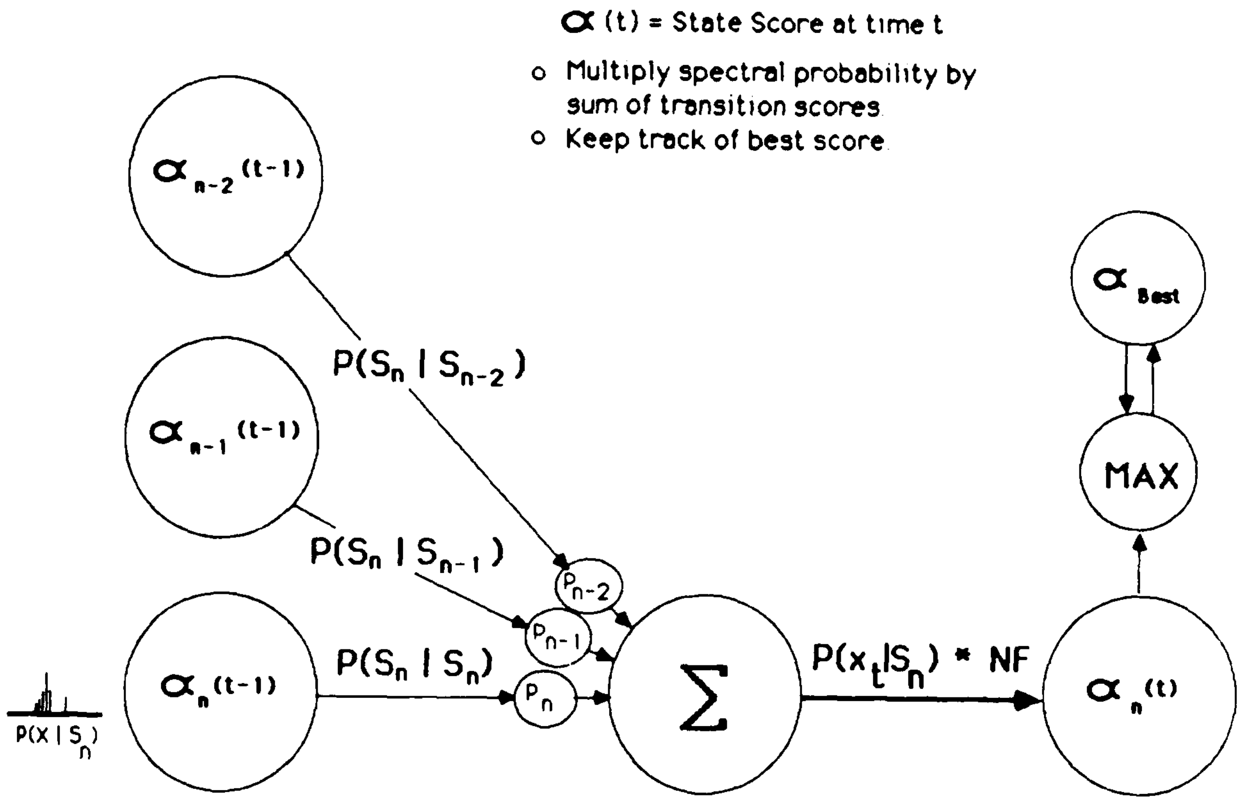

Figure 1: Score Computation for One State

```
h)          replace max score
i)        IF (new max terminal)
j)          replace max terminal
k)    determine normalization
l)    FOR all words
m)        get terminal score
n) determine theory
```

The first parallel version combined lines d) through f) and parts of g) and h) into a single task and used the generator GenOnIndex, which includes a prologue task and an epilogue task in addition to the main task. The prologue task is executed only once by each processor before that processor executes the main task for the first time. In this version, the prologue included line b). Similarly, the epilogue task is executed once by each processor after all main tasks have been completed by that processor. For this program, the central task determined the maximum state score and the maximum terminal score seen by each processor. The epilogue task compared these local maxima against global maxima, replacing the global maxima if necessary. The remainder of the program (lines k-m), including the second FOR loop was executed sequentially, on a single processor. Approximately 9 seconds was spent in the sequential portion of the program when it was run for a 3.5 second utterance. When run on 15 processors, this resulted in less than 50% utilization of the processors.

The next step was to attempt to reduce the sequential portion of the program. We noticed that the second FOR loop (lines l and m), which propagates the best word's score in the current frame to word-initial states for the next frame, could be incorporated into the first FOR loop, effectively changing the program to:

```
FOR all frames
  initialize frame
  FOR all words
    get previous frame terminal score
    initialize word
    FOR all states
```

```
          compute state score
          IF (new max score)
             replace max score
       IF (new max terminal)
          replace terminal
    determine normalization
determine utterance.
```

This revision substantially reduced the time spent executing serial code. For 15 processors, the execution time dropped from 15 seconds to 11 seconds for a 3.5 second utterance, and the effective number of processors rose from 6.9 to 11.2, or approximately 75% utilization.

2.3 Large-Scale Parallelism

In this section, we describe the work needed to make the recognition algorithm efficient on a larger scale parallel processor. While the initial algorithm was 75% efficient on a 16-processor machine, the efficiency dropped to 20% when a 97-processor machine was used. Although this represents a factor of 20 speedup of the program, it is an inefficient use of the machine. This section presents several factors that contributed to the inefficiency as well as the methods used to improve them.

As Amdahl's law states, when the number of processors becomes large, any serial computation in the algorithm will quickly dominate the computation time, thus reducing efficiency. We identified a number of problems:

- too few tasks for each processor
- scheduling overhead was too large
- uneven length tasks caused some processors to be idle while waiting for frame synchronization
- computation of the maximum score was a serial process.

As described in the rest of this section, we found solutions to these problems. Briefly, they were:

- Attack problems of sufficient size for a parallel processor.

- Use a more efficient scheduling algorithm.
- Perform tasks in decreasing order of length to decrease the amount of time spent waiting for synchronization. In addition, the algorithm was changed so that some work from the next frame could be performed before synchronization, thus keeping processors busy when they had finished the work on the queue for the frame.
- Implement an algorithm that performed the maximum in logarithmic time.

With these changes, the efficiency improved from 75% to 95% on a 16-processor machine and from 20% to 79% on a 97-processor machine.

There are a number of potential obstacles to attaining efficient processor utilization on a multiprocessor. Typical issues include contention for a common memory location, serial code in the program, and processors waiting idly to synchronize with other processors. Each of the specific problems described below includes one or more of these issues.

Number of Tasks and Startup Overhead

Even before the program was run on a larger machine, we had anticipated that it would be hard to obtain high processor utilization with a vocabulary as small as 120 words. Since our long-term goal is to recognize speech from large vocabularies, we switched to a larger task of 335 words. This change improved processor utilization to 35% on the 97-processor machine.

The speed of processor scheduling was examined next. In the initial parallel version shown above, the generator subroutine call starts all the processors at each frame. It was found that the overhead of starting up was relatively large for the amount of work being done at each frame. To reduce the overhead, the program was altered to start all processors only once at utterance start, generating $Nframes \times NWords$ tasks at that point and letting each processor determine its word and frame indices from the single task index it receives from the generator. Processor utilization improved to about 50% with this change.

In changing the program to start processors only at the beginning of an utterance, we had implemented an explicit synchronization to replace the implicit one provided by starting up a new generator at each frame. We subsequently made two more changes to synchronization to improve efficiency.

Processor Synchronization Issues

The task generation change had removed the synchronization provided by starting up a new generator at each frame. To replace this, an explicit synchronization was built into the program to be performed after all the words in a frame were processed. There were two subsequent changes to improve the efficiency of synchronization. The first dealt with task ordering. In the early versions of the algorithm, processors updated *all* the words in the vocabulary, with no particular ordering of the words. Since words have varying numbers of phonemes (from one to 14 phonemes in this task's vocabulary), different words took different amounts of time to update. If a processor began work on a long word near the end of the work for a frame, other processors would finish their assigned words and wait idly to synchronize with the one busy processor. To reduce this inefficiency, the words were processed in order from longest to shortest (in number of phonemes).

In figure 2, we schematically depict the situation before and after the words are ordered. The filled rectangles represent time when processors actively work on tasks and the white space represents time between tasks when no work is being accomplished. In the right hand part of the figure, idle processor time is substantially reduced by sorting.

The second change to synchronization efficiency concerned the point in the program at which synchronization was done. As mentioned, the purpose of the synchronization was to ensure that no processor proceeded to the next frame until the starting score for words and the normalization factor were computed. Since the normalization factor was only to avoid score underflow, it could be estimated a frame or more earlier. The only remaining synchronization constraint was the word-starting score. This score, however, is used only at the beginning of the *first* phoneme of each word. Considering this, the order of the update of a word was reversed so that the last phoneme was updated first, and the first phoneme updated last. This change allowed a processor to finish work on one frame and immediately begin work on updating a word from the next frame, synchronizing only when it got to the first phoneme. In this way, time that had been previously spent by processors waiting for others to finish a frame was now being used to perform useful work from the next frame.

Figure 3 depicts the situation for two frames of an utterance before and after this change. Tasks for time T+1 are shown in two shades. The darker portion represents the part of the task that depends on the previous frame's work being finished. On the right, with the order of the computation reversed, the idle processor time is reduced. The effect of the synchronization changes was to increase processor utilization to approximately 72%.

Finding Global Maximum

Finally, the efficiency of finding the maximum value was also improved. A straightforward computation of the maximum value requires that all values be compared with a single memory location, but this approach results in contention for that location. As a first improvement, the program was altered to make each processor maintain its own *local* maximum of the scores of all the words that it updates in a frame. At the end of the frame, the global maximum of these

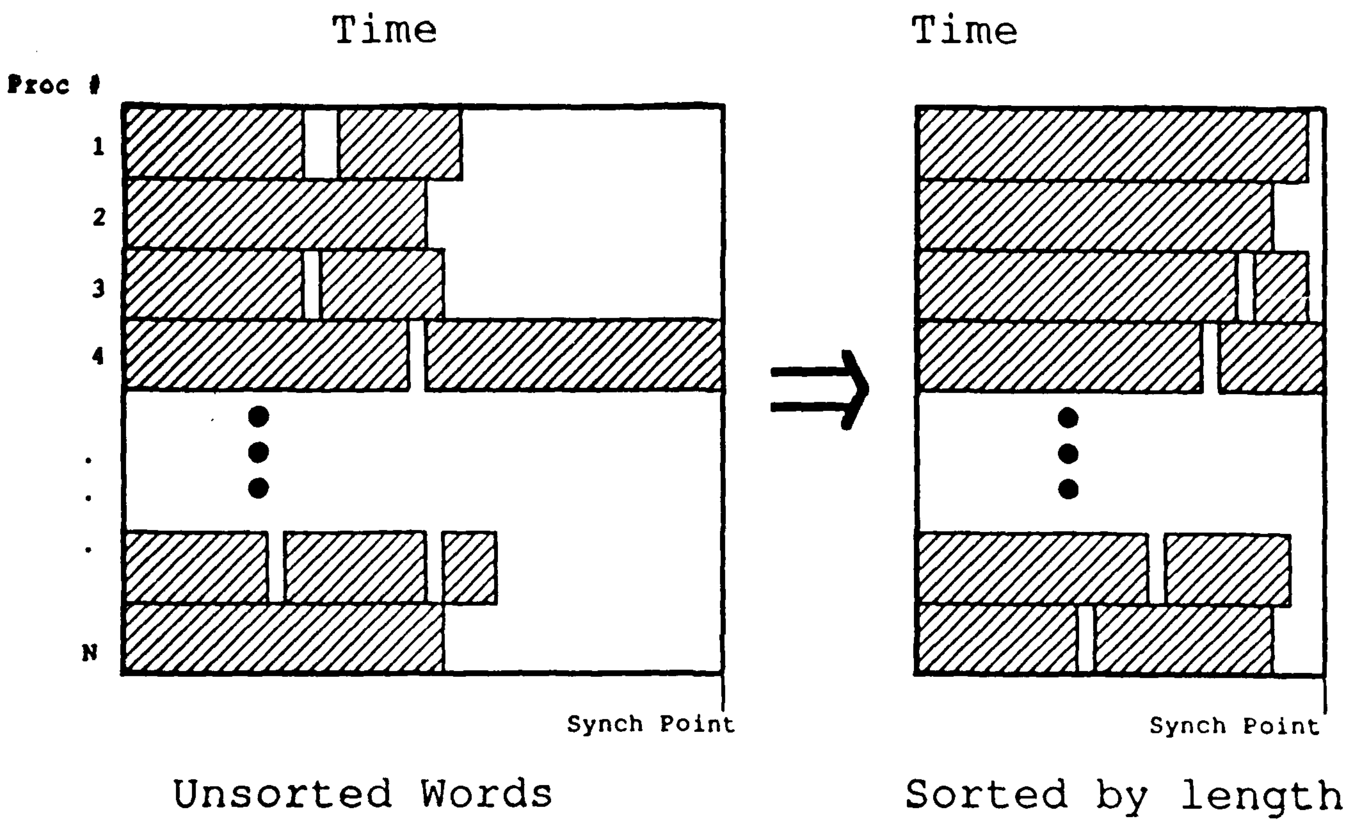

Figure 2: Ordering Tasks by Length to Achieve Load Balancing

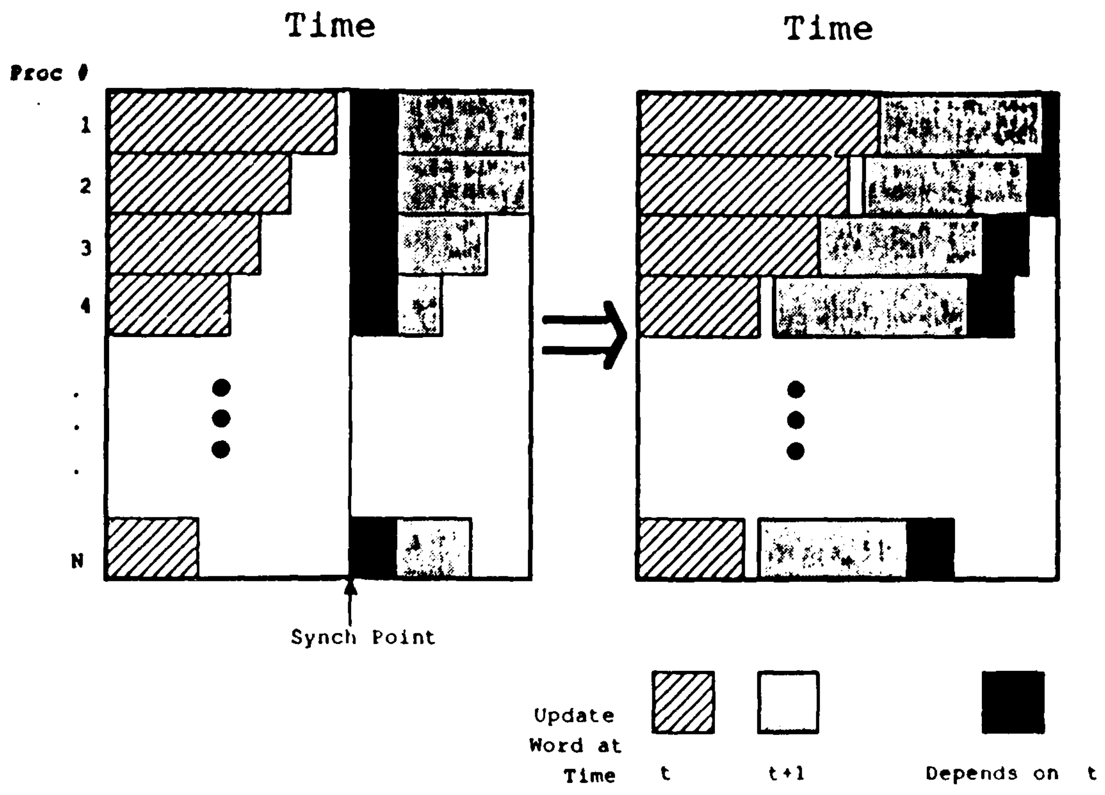

Figure 3: Reversing Word Computation Order to Reduce Synchronization Waiting

values over all processors was determined. In initial versions, this was accomplished by having processors sequentially compare their value to the global location and replace it if necessary. In taking the global maximum, each processor's operations must be done indivisibly, since processors could otherwise interleave operations and get inconsistent results. In early versions of the program, this was accomplished, by using "locks" and having processors sequentially compare their local maximum to a global location. Although on a sixteen processor machine, the time for processors to turn in values in this way is negligible, with 97 processors, the inefficiency of the approach becomes noticeable.

An alternative to this approach was to set up a "binary tree" of locations for taking the maximum. In this approach, the processors' local maxima are the leaves of the tree and the maxima are propagated up through the nodes of the tree. This approach reduces the asymptotic time for finding a global maximum from O(N) to O(log N), where N is the number of processors. More importantly in our case, efficiency improved because memory contention was reduced. The inefficiency might be reduced further by increasing the branching factor of the tree (e.g. having four processors turn in their local maxima to a location instead of two). However, since the loss was determined to be small, this approach was not pursued further.

The total effect of all the improvements described above was to improve processor utilization on a 97-processor machine from 20% to 79%. Figure 4 is a graph of processor utilization for 1 to 97 processors on the 335 word task. The actual speed of the speech recognition improved accordingly. After the optimizations are included, a one-processor Butterfly Parallel Processor requires 128 times real time (128 seconds to process one second of input speech) and a 97-processor machine requires about 1.7 times real time. It should be noted that these times are for Butterfly Parallel Processors that use MC68000 microprocessors. When we used the later version of the Butterfly that used the MC68020 based machine, the machine sped up by about 60% to achieve real time.

2.4 Speech Recognition with a Grammar

Speech recognition with a grammar presents special problems for a parallel implementation. In the no grammar case, the parallelism is very regular: there is a fixed number of words whose scores are updated in parallel, with a single global maximum at the end of each frame. With a grammar, there are several sets of words, with each set coming into a different node in the grammar. Each of these sets of words must be handled independently, making the parallel algorithm much more complex. Typically only a small percentage of the possible words in the grammar score well enough to be updated. Therefore, the scheduling algorithm must use a dynamically changing list of words to update rather than just updating all words. Furthermore,

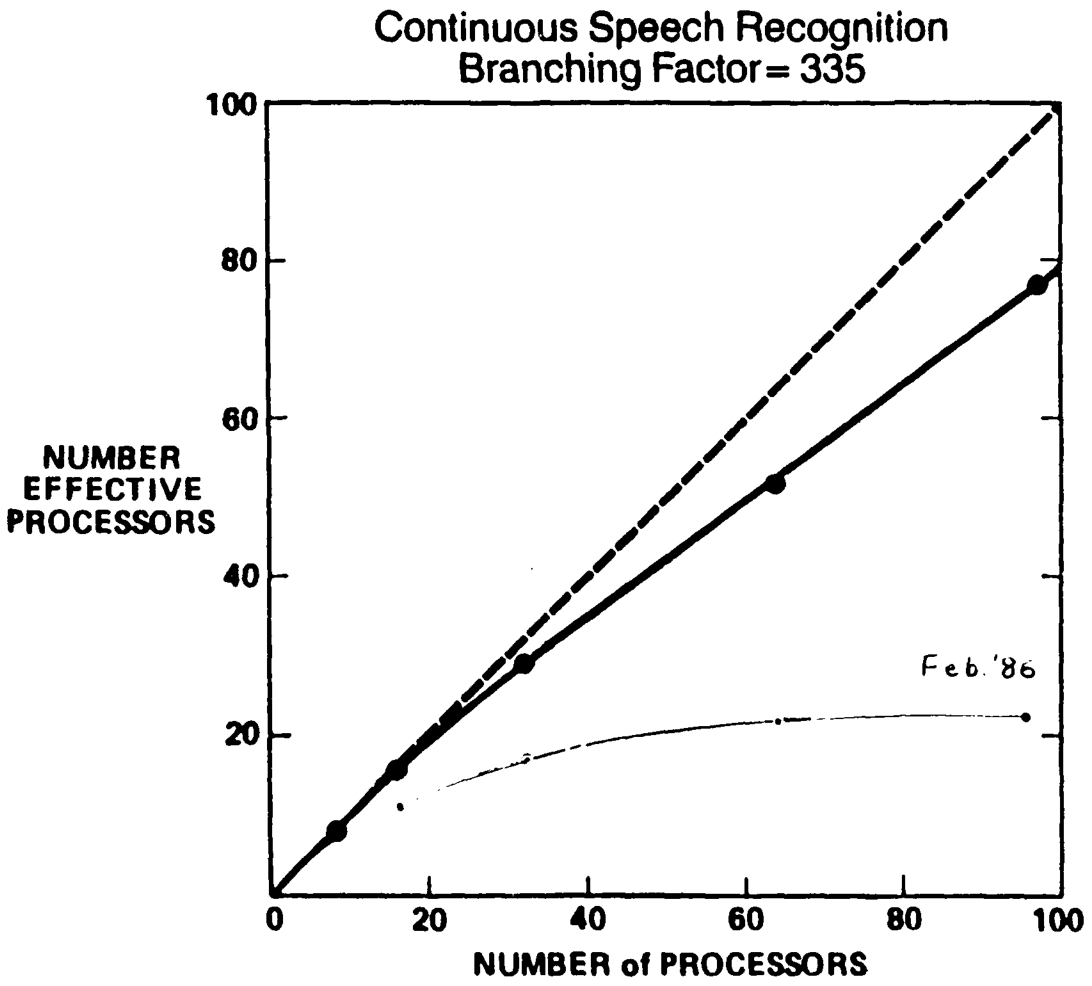

Figure 4: Butterfly Processor Utilization, 335 words

the amount of work varies significantly from frame to frame. As a result, in frames with very little work to do, some processors will not have enough work for efficient parallel utilization. Below we discuss the algorithms implemented to recognize speech using different grammars and to deal with these special problems.

Pruning

One issue for grammar recognition is that only a small percentage of words actually score well enough to be worth updating. In a serial machine, a simple pruning algorithm can be used to avoid some computations. For the case of no grammar, pruning within the word typically saves a factor of two in computation. However, for a parallel machine, reducing the work in this way may reduce efficiency. Before any grammar recognition algorithms were developed, we implemented within-word pruning. We found that this algorithm saved the same factor of two as the serial implementation, even though the tasks were now shorter and more varied in length. With this change, the system could perform recognition with a vocabulary of 350 words in 1.2 times real time on a 64-processor machine. That is, each second of input speech required 1.2 seconds of computation time.

Bigram Grammar

The first type of grammar that we implemented was a statistical bigram grammar. This grammar has a probability for each pair of words in the vocabulary. That is, given a preceding word, there is a different probability for each following word. This type of grammar can provide good recognition accuracy with a relatively simple algorithm. In the initial implementation of this grammar, word-ending scores that were sufficiently high were propagated forward to all possible words that could follow. This approach had the drawback that it required using memory locks to guarantee that the maximum was computed correctly. An alternative method was to maintain a list of good ending words and have each new word look back at this list to determine its starting score. This approach did not require locks and therefore reduced serial contention. The recognition speed was similar to that for no grammar: it was 6 times real time on a 16-processor machine.

1000 Word Vocabulary

At this time we changed task domains to the 1000-word DARPA resource management task. The increased vocabulary size required increased memory in the decoder. At this point, the Butterfly had a 16 MB virtual address limit. We compressed the decoder's data structures in order to fit them in the limited space. From this point on, all work used this task domain.

DFA Grammars

The second type of grammar that we implemented was a Deterministic Finite Automaton (DFA). This grammar is a directed network of nodes and labeled arcs specifying all possible word sequences. At any node of the DFA grammar, the exiting arcs of the grammar contain at most one instance of each word. This type of grammar is useful for specifying small but useful languages.

The first major change to the algorithm was that we implemented a new scheduler that maintains a list of the grammmar-word-arcs that are worth updating. This list was implemented as a first-in first-out queue, where new tasks were added to the tail of the queue and processor took their tasks from the head of the queue.

To decode a sentence, the processors updating the word-arcs deposit information (ending score, maximum score, traceback) into the arc's destination node. Whichever processor updates the last arc into a node is also responsible for doing "node work". This entails propagating the best ending score into all legal following words, specified by the exiting arcs from that node. It also will schedule tasks for each of these words that are not already on the queue for the next frame. Finally, the maximum score from all the words' arcs entering the node is deposited into a global array. Similarly, the processor that does the node work for the last node is responsible for doing the "frame work". That is, it finds the maximum of the scores in the global array, computes the normalization factor and pruning threshold for the next frame, and starts the processing for the next frame. At this point any processors that have been waiting for the current frame to end can proceed to the next frame. After the work for the last frame is completed, one processor traces back to find the best scoring sequence of words for the utterance. This traceback is done by a single processor, but is computationally insignificant enough that it was not worth parallelizing.

NFA Grammars

The next type of grammar that we implemented was a Non-deterministic Finite Automaton (NFA). This is a more general type of finite automaton in that non-unique exiting arcs are allowed from a single node. In addition, it allows a special type of arc called null-arcs. Null-arcs allow transitions from one node to another with no words between them. This makes it easy to represent the possibility of skipping a word or several words.

The advantage of NFAs is that they can represent the same languages as DFAs in a more compact form. In addition, they often require less computation in recognition. These advantages make it practical to use more complex language models, such as the ones described below. However, NFAs are more difficult to implement than DFAs since null-arcs introduce an additional part of the computation that must be done after the word updates are finished in a frame. Efficient algorithms for propagating scores through null-arcs in parallel require some overhead to avoid processor contention. In addition, the null-arc score propagation involves different null-arcs at each frame and it can be scheduled only after the word updates are done.

We completed initial work on NFAs and implemented the system with a number of different grammars. One of these grammars was used in the Speech and Natural Language Project's initial "serial demonstration" described below. Using this grammar, the system recognized utterances in approximately 3-5 times real time.

Word-Pair Grammar

For our live test on September 29th and for the demonstration on October 13th, we wanted to use a grammar with large perplexity that also covered all of the sentences in the database. A simple grammar that met both of these goals is the Word-Pair (WP) Grammar. This grammar allows all of the pairs of words that are allowed by the set of sentence patterns that define the DARPA Resource Management Task Domain Database. On the average, for each of the 1000 words there are about 60 words that can follow, i.e., the grammar perplexity is 60.

The WP Grammar can be implemented as an NFA that has 1000 arcs for the 1000 words in the vocabulary, and null-arcs connecting all allowable pairs of words. Thus, this grammar has a very high percentage of null-arcs compared to many other grammars. Using the implementation described above for NFAs, the time required for this grammar was about 60-80 times real time on 16 processors. To avoid duplication of work, the initial implementation required that each word that was scheduled to be updated notify all nodes that could be reached directly or through null-arcs by that word. For the WP Grammar, this notification was very expensive because of the large number of null-arcs. In fact very few words end with good scores. Therefore, we were able to remove the notification step by adding a flag so that only the first word to propagate to a node would schedule following word tasks. This new algorithm reduced the computation to 10-15 times real time with an efficiency of 47%.

At this time we still felt that the recognition should run about twice as fast as it was running. We used a new profiling tool that allowed us to monitor the different tasks in the parallel processor. This tool showed clearly that, in each frame, all processors worked on each of the words to be updated, and then a few processors propagated the node scores through the null-arcs to the possible following words. (The WP grammar is somewhat different from other grammars in that it has a very high ratio of null-arcs to word arcs.) We modified the algorithm so that rather than requiring the last processor into a node to propagate scores from that node, these tasks were put on a second queue. This second queue of tasks was executed in parallel when all of the first queue was completed. Thus, work that was executed in serial was now being executed in parallel. The efficiency improved to 68%, and the speed improved to 6-8 times real time or 10-30 seconds for average-length utterances.

Overall Strategy

In all of the code implemented on the Butterfly, we tried to avoid locking memory where possible. Instead, we used atomic operations to increment and decrement counters to assure orderly updating of values. When necessary, we used even/odd registers or last-frame/current-frame/next-frame registers to avoid confusion between different frames. This was necessary because we allowed the Butterfly to be working on two frames at one time. While we were able to increase the parallel efficiency by working on two frames at a time, it did complicate the algorithm considerably, especially in combination with grammars and pruning and attempting not to use locks. There were frequently race conditions that had to be debugged. These were usually related to working on two frames at a time.

In conclusion, we believe that it is better to avoid complexity in the algorithm if we have a choice. This would make using the Butterfly much easier, as long as we don't require efficiency very close to 1. One approach would be to reevaluate the need for each part of the algorithm depending on the condition. For example, it is worth working on two frames at once when the number of word updates per processor per frame is small (less than 10). In this case, the cost of having all but one processor wait for one to finish is substantial. (Note that for real time on the Butterfly, each processor can only work on 3 words per frame.) For large tasks or small Butterfly configurations, it is probably not worth the complexity, since the percentage of time lost due to waiting is smaller.

3. Demonstrations

During the third year of this contract, we gave several demonstrations of speech recognition and understanding using the Butterfly implementation of our continuous speech recognition system. The first, on July 27th, demonstrated a serial connection between speech recognition and natural language understanding within a simulated IDB/OSGP task. IDB is an iteractive database system for accessing the FCCBMP database. OSGP is an interactive display facility for displaying charts, ships, sail-plans, etc. We implemented a simulation of a subset of these two systems in order to demonstrate the feasibility of real-time interactive access to the database and expert systems through the use of speech and natural language. The demonstration was accomplished by connecting a VAX, two LISP machines, and the Butterfly (through a UNIX VAX). The grammar used was a subset of the grammar for the whole task domain, designed to cover all sentences related to the demonstration scenario. The perplexity of the grammar was 40, measured on a test set of sentences. This indicates that the grammar represented a fairly difficult recognition task. The system included an interactive graphics capability in which the user could display charts of different oceans, control the display of different ships and their tracks, query about the readiness of different vessels, etc. The speech recogition for each sentence required about 10 seconds for each sentence. This was followed by about 20-30 seconds of time on a Symbolics Lisp machine interpreting the meaning of the recognized words, followed by a rapid response on the graphics screen or answer from the database query that was generated. While the machine interaction was complex, the demonstration gave a flavor for how a complete spoken language system might appear.

The next demonstration using the Butterfly was the Live Test performed on September 29, 1987. In this case, each of three speakers, Allen Sears, Dave Pallett, and Tice DeYoung, spoke 30 sentences directly into the Butterfly recognizer. The grammar used in this case was the Word-Pair Grammar described in the previous chapter. This grammar has a perplexity of about 60. The speakers were each given a list of sentences chosen by Dave Pallett. The speaker would read each sentence, wait for the system to type out the recognized answer, and then after the prompt, read the next sentence. Each sentence typically required about 20-30 seconds for decoding (about 10 times real time) on a 32-processor Butterfly. Each speaker had an allotted time of 30 minutes to process all 30 sentences. Even with initial time for setting up and discussion between sentences, all speakers were able to finish within the 30 minute time limit. The results of these live tests were presented at the DARPA meeting on October 13, 1987. The word error rates for the three speakers were 4%, 5%, and 12%, respectively.

On October 13th, we demonstrated the Butterfly recognition system at the DARPA speech meeting held at BBN. The system had been enhanced with graphics capabilities to illustrate the recognition process as it was happening. First, the best partial answer was displayed as it was computed. At the end of each frame, the best theory was examined. A full traceback was performed to determine the sequence of words in the best theory. Then, if this sequence was different from the sequence that had been displayed before, the preceding sequence was erased and the new sequence was displayed. Since the display process was quite fast, it appeared as if words were being added one by one from left to right. If the answer changed, it would change the string accordingly. Typically, a few words from the beginning of each utterance were displayed before the utterance had been fully spoken, indicating that the recognition process was taking place at the same time as the utterance was spoken.

The number of active word-arcs in each frame was displayed. For the word-pair grammar, the maximum number of active word-arcs is the vocabulary size. Therefore the displayed number varied from a minimum of one to a maximum of the vocabulary size. For other grammars, the maximum number of word-arcs could be more or less than the vocabulary size. The number of active word arcs fluctuates considerably due to two factors: the number of words that can follow each word in the part of the sentence recognized so far, and the acoustic uncertainty at each part of the utterance. This variation in the amount of work makes it necessary to have a dynamic allocation of tasks to the different processors in a parallel machine.

Finally, the work load balancing was illustrated by showing the number of phonemes updated by each processor. Each processor kept track of the number of phoneme models that had to be updated. After every quarter second, the totals were displayed as a bar diagram, with one bar for each processor. This showed that, on the average, all the processors were performing about the same amount of work.

4. Board-Level Recognizer

Since the aim of this project is real-time speech recognition, we have also been considering the use of array processor boards as a small, portable alternative to the Butterfly for demonstration purposes. For this task, we require a board that is not only powerful in terms of computations, but has very large memory required by the recognition problem. We considered two very different options. The first was to buy a commercially available array processor board. The second was to build a special-purpose board tailored for speech recognition.

4.1 Commercially Available Boards

We considered several commercially available boards built by Sky, Mercury, and BBN Delta Graphics, in addition to others. The conclusion for all of these boards was that they either did not have sufficient on board memory, or were not fast enough for real-time operation. We estimated that the speed would be comparable to that of a moderately sized Butterfly.

4.2 Special Purpose Recognition Boards

After discussions with Hy Murveit of SRI, we decided that it would be most interesting to build a special purpose board, because the speeds that could be attained would be at least an order of magnitude faster than could be attained on any commercially available general purpose board. In addition, Murveit believed that it would be possible to complete the design and fabrication of a prototype in under one calendar year. The effort to build a prototype special-purpose speech recognition system began around November of 1987. We were involved in three different phases of this effort:

1. Design of the high level hardware architecture to accomodate the algorithm
2. VLSI design of some components
3. Front end signal processing software on TMS320C25

Each of these is described below.

We agreed to cooperate fully with SRI and with Bob Brodersen at UC Berkeley to help in the design and implementation of a special purpose recognition board capable of handling a class of discrete HMM recognition algorithms. We attended meetings held at Berkeley and at SRI to explain the details of our HMM recognition algorithm, to discuss the computational tradeoffs of different ways of structuring the algorithm, and to begin in the hardware architecture. Through these discussions, we arrived at a design that was an order of magnitude faster than the initial design.

A VLSI designer from BBN spent several months at UC Berkeley to assist in the design of some of the components. This involved learning the design tools and the methods used there.

We specified the hardware and software requirements for the front end signal processing board. These requirements were documented explicitly and distributed to SRI and to UC Berkeley. The signal processing board was designed to have two TMS320C25 integer array processor chips, each with 64K bytes of static RAM. One of the processors has an A/D and filter chip attached to it. The two processors are connected via a serial interface. Finally, each of the memories is part of the address space of the 68020 host. We implemented the signal processing algorithms that we currently use in integer arithmetic on the dual TMS320C25 system. All of the software was tested by running actual speech data through it and comparing the results to floating point results obtained on the FPS array processor. We included the necessary scaling operations in the program to ensure that the differences were minimal. At the present time the software is complete and has been tested in the simulator, so our part of this work is complete.