Interpreting Noisy Prose using Heterogeneous Multi-cores

Jerry Potter  
Department of Electrical and Computer Engineering  
Colorado State University  
Fort Collins, CO 80523-1373 USA

Howard Jay Siegel  
Department of Electrical and Computer Engineering  
and Department of Computer Science  
Colorado State University  
Fort Collins, CO 80523-1373 USA

Abstract - Currently, verbally dictating commands and programs to a computer is not practical because in our experiments, 10% to 15% of the words and 30% to 80% of the commands and statements were misrecognized by the Automatic Speech Recognition (ASR) systems. Interpreting Noisy Prose (INP) is a post ASR algorithm designed to execute on and utilize the extensive computing power of heterogeneous multi-core chips. The algorithm generates several variations of an ASR’s output based on common and user defined homonyms, misunderstandings and “near homonyms.” The variations are compared against domain specific grammars and lexicons. To achieve acceptable speeds and accuracy, INP uses parallel computation to follow the branches of a search tree and data parallel matching at nodes of the tree to evaluate the sets of alternatives and select the “best” interpretation of the ASR’s output. The initial INP results were 99% correct recognition for words and 94% correct recognition for commands. The heterogeneous approach reduces the number of parallel tasks by about a factor of 150.

Keywords: heterogeneous, multi-core, prose programming, SIMD, speech recognition.

1 Introduction

The real estate of portable digital devices and remote controls is precious. Few such devices have room for the ever growing number of buttons, switches and displays. Using screens to display icons and menus is an inefficient use of a critical resource because it limits what can be input and interferes with visual output. There is a need for more sophisticated methods of communication between people and their computers and digital controls.

Natural language has been the method of choice for human communication for hundreds of thousands of years and is an obvious choice for a real estate limited interface. Plain English [1] is a high level pseudo prose programming language that can be typed or spoken. It supports the associative model of programming [2] and with adequate voice recognition is well suited for programming databases and spreadsheets, searching the web and controlling real estate limited handhelds such as cell phones, remote controls, PDAs, and PCs.

Voice command systems, such as Microsoft’s Voice Command for Windows Mobile, e-Speaking, Fieldsoft’s AIMSonScene Incident Command System are being used more and more frequently. Fonix Speech recently announced the VoiceIn™ (Nov. 2008) command software for Tom Clancy’s “Endwar™” Video Game. Unfortunately, Automatic Speech Recognition (ASR) output is often “noisy” or ambiguous and such systems frequently limit their domains to achieve acceptable rates of speed and accuracy. In the Plain English environment where the domain is not limited a priori (commands may be add or modified by the user), modern voice recognition systems typically only recognize on the order of 90% of the words spoken [3].

The Interpreting Noisy Prose (INP) post ASR algorithm was able to improve our “off the shelf” ASR system’s recognition rate from 85% to 90% for words and 20% to 70% for commands to 99% for words and 94% for commands. Typical ASR post processing approaches use lexical statistical methods, voting and minimum edit distance of corrections based on domain knowledge, and morphological and query template information. They obtain word accuracy in the high 80% [4,5,6]. In contrast, INP generates numerous alternative versions of the ASR output based on known homonyms, near homonyms and misrecognitions and then uses the massive parallelism available on multi-core chips to select the alternative that best matches the domain dependent lexicons and command patterns. The parallel INP algorithm was developed and simulated on a conventional PC using a conventional voice recognition system. It is estimated that the algorithm can reduce the number of misrecognized words and commands to levels approaching that acceptable for every day use.

This paper describes the parallel multi-core INP algorithm. INP is a post ASR algorithm designed for heterogeneous multi-core chips that significantly
improves speech recognition speed and success rates to a level suitable for prose programming and device control. First, in Section 2, a brief overview of Plain English is given. Section 3 defines terms and data structures. Section 4 is a description of the INP algorithm. Section 5 summarizes the results. Section 6 gives the conclusions and outlines future research.

2 Plain English

Plain English and the associative calculus has been described elsewhere [1,7], but it is helpful to review some of the basic concepts here. The associative calculus is based on representing data in spreadsheet like tables with row and column names. See Figure 1. The calculus asserts that data can be accessed in data parallel form simply by using a column’s name (e.g., overtime). Rows are equivalent to records and may be labeled (e.g., Joan). If the rows are labeled, individual cells in the table can be accessed by specifying their row and column labels (e.g., Joan’s overtime). Thus the calculus alleviates the need for structure declarations and the user defined access functions used frequently in C++ and other high level languages.

<table>
<thead>
<tr>
<th>Payroll</th>
<th>overtime</th>
<th>rate</th>
<th>overtime wages</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>20</td>
<td>8.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joan</td>
<td>17</td>
<td>10.34</td>
<td></td>
<td>50.00</td>
</tr>
</tbody>
</table>

Figure 1 – An Associative Calculus Table

```
for (i=0; i< number_of_employees; i++)
  overtime_wages[i] =
                  overtime[i] * overtime_rate[i] + special_bonus;
```

a – An Innermost C++ Loop

Multiply the overtime by the overtime rates.
Add the special bonus.
Put the result in overtime wages.

b- Plain English Data Parallel Statements

Figure 2 – Contrasting C++ and Plain English

```
ADD arg1 dot
MULTIPLY arg1 BY arg2 dot
PUT arg1 IN arg2 dot
```

Figure 3 – Sample Plain English Grammatical Patterns

Plain English uses case grammars, associative calculus, and pronoun references to effect pseudo prose statements and commands that are easy to learn, use, remember and vocalize. Plain English also uses the similarity between the natural language singular/plural dichotomy and the computer science scalar/data parallel dichotomy to eliminate innermost loops. See Figure 2. The case grammatical patterns for the Plain English statements in Figure 2b are shown in Figure 3. The upper case words are terminal symbols (keywords). “dot” is a period as opposed to a decimal point.

The INP algorithm uses case grammar patterns [8] and pattern matching to utilize the parallelism on multi-core chips. The interpretation of a Plain English input sequence depends on recognizing a verb and its prepositions. These items are used as keywords for selecting one or more appropriate grammatical patterns. The selected grammatical patterns are used to parse the input into phrase segments. Data parallel pattern recognition techniques are then used to match the phrase segments against the lexicons, which map them onto table, row and column number triples. These triples are combined into associative, data parallel, assembly language level code [2]. The assembly level code is general purpose and can initiate all conventional digital device activities both external, such as command and control actions, and internal, such as spreadsheet manipulation and database creation and update.

3 Definitions and Data Structures

**Sequences** – Sequence is used as a general term for the noisy ASR output that is the original input to the INP algorithm. It is a sentence, command or question with possibly misrecognized words. That is, a sequence is the potentially noisy prose input.

**Segments** – Segment is used to refer to the parsed portions of the sequences. Segments correspond to potential verb and prepositional phrases.

**PVs and TRCCs** – There are two types of segments – PVs and TRCCs. **PV** stands for Preposition or Verb. A PV segment consists of the possibly multword verb or preposition alone. ADD, MULTIPLY, PUT, BY and IN are the PVs in Figure 3. **TRCC** stands for Table, Row, Column or Cell identifier. TRCCs are essentially the noun phrase portion of a PV phrase and may contain modifiers as well as a noun. The cell name “special bonus” and the column name “overtime wages” are examples of TRCCs from Figure 1.

**Homonymization** - ASR output may contain homonyms of the actual intended word. For example, the homonyms “to,” “too” and “two” are often confused. Thus an ASP program might interpret the verbal command “Add 2 to 2 ." as “add to two too." For input sequences such as this, all possible combinations of the homonyms are generated and processed (including the possibly correct original sequence). The process of generating these combinations is called homonymization and results in “correcting” misspellings, misrecogsitions, mis-
typings, etc. The results of the homonymization process are called *homs*. Specifically, homonymized sequences, wherein only verbs and prepositions are homonymized, are called *PV homs* and homonymized TRCC segments, wherein nouns are homonymized, are called *TRCC homs*.

**Grammatical patterns** – The grammatical patterns used in INP are based on case grammars [8] and are used primarily to divide the input sequence and its PV homs into PV and TRCC segments. For example, in the grammatical pattern to recognize an arithmetic multiply operation shown in Figure 3, MULTIPLY and BY are verb and preposition terminal symbols and arg1 and arg2 are non-terminal symbols that specify a specific class of nouns that will be allowed.

**Slots** – A slot [9] is a data structure that contains all of the information pertaining to a PV phrase. There is one slot for every case [8] in a sequence.

**Frames** – A frame [9] is a collection of slots that together contain all of the information pertaining to a sequence, i.e., the input command or statement.

**Environment** – Most of the information for the INP algorithm is stored in tables. An environment’s database consists of one or more tables. The tables are named, and the tables’ rows and columns may be named. These names constitute the nouns of the TRCC homs. Names may be composed of multiple words.

**Domains** – The verb and prepositions of the grammatical patterns, the nouns formed by the table, row and column names, and the contents of the tables themselves constitute the different domains of the environments.

**Lexicons** – A lexicon is a listing of all the possible noun phrases in an environment’s database table. Lexicons map the nouns of the TRCC segments onto table, row, column and cell address combinations. Referring to the lexicon for the payroll table of Figure 1: “Jane’s payroll overtime” maps onto (0,0,0), “Joan’s overtime” maps onto (0,1,0) and “special bonus” maps onto (0,2,3); where (x, y, z) represents table x, row y and column z and numbering starts at zero. If not explicitly stated, the table and column addresses default to the last specified values.

**Understanding** – An input sequence is understood by finding a set of records in the environment that is best identified by the input sequence. The set of records can be used to generate different outputs such as associative computer assembly language instructions, web search instructions, database and spreadsheet instructions, and digital device control instructions.

**Parallelism** – The INP algorithm uses the two general types of parallelism as defined by Flynn [10] and described in Hennessy and Patterson [11]. *MIMD* (Multiple Instruction stream, Multiple Data stream) refers to control parallelism or any type of parallelism that uses separate instruction streams for every processing unit. *SIMD* (Single Instruction stream, Single Data stream) parallelism refers to data parallelism [12], wherein one instruction stream is shared by or broadcast to numerous, i.e. hundreds, thousands, even hundreds of thousands, of processing units that execute it simultaneously. The numerous SIMD processor units may be in one core or in multiple cores. *Associative parallelism* [2] is a type of SIMD parallelism that uses flags and tags to partially record the state of a computation on a per datum or record basis and thus associate the results of one data parallel operation with another. *Multiple associative parallelism* [13] refers to a type of associative parallelism where instructions from multiple instruction streams are sent in parallel (as per a VLIW) to all processors of a SIMD core or computer and the processors select the instruction to execute based on the per datum flags and tags.

Both MIMD and SIMD computers are general purpose and can execute sequential code as well as emulate any kind of parallelism. However, it is assumed that they execute their own variety of parallelism most efficiently. Specifically, note that *SPMD* (Single Program, Multiple Data) parallelism [14] is often used on MIMD machines to simulate SIMD parallelism; but it is not SIMD parallelism per se because the single program (or its threads) is replicated many times using many instruction streams incurring substantial hardware and software overhead costs for control, synchronization and communication [15, 16].

When both MIMD and SIMD cores are present on a heterogeneous multi-core chip, the MIMD cores can be programmed using conventional techniques such as Open MPI. When a MIMD core “controls” one or more data parallel (SIMD) cores (that is, when one or more of the MIMD cores execute the sequential program and associated operating system software and send the data parallel instructions to one or more SIMD core “slaves”), the MIMD/SIMD configuration is operating in *multiple SIMD* mode.

In is important to note that from the Plain English programmer’s point of view, SIMDs are the basic compute engines and data parallelism the basic mode of computation. Scalar computation is simply a trivial form of data parallel computation. The MIMD cores only assign tasks, calculate parameters, control synchronization and I/O, etc. for the SIMDs’ benefit.

### 4 The INP Algorithm

In noisy Plain English input sequences, the verb and preposition keywords that are the grammatical pattern terminal symbols may be, and often are, “garbled” or misrecognized. Consequently, the input may be matched by several different grammatical patterns resulting in the
initial multiple paths in the search tree. The basic approach of the INP algorithm is to use spatial multi-tasking MIMD parallelism to follow the various branches of the tree and to use data parallel SIMD parallelism, in multiple SIMD mode, for matching patterns at the nodes of the tree. The alternative branches and matches are scored and evaluated to select the “best” interpretation of the input.

Figure 4 illustrates the MIMD parallel fan out and the interaction with SIMD parallelism. The SIMD tasks are represented by rectangular boxes.

First, the initial input sequence is homonymized with several verb and/or preposition corrections. For example, the top level sequence “bad age two year” may be homonymized to “add age two year,” “bad age to year,” “bad age too year,” “add age to year,” and “add age too year.” Based on our experience with the commands in Appendix A, three or four (say $m$) PV homs are commonly produced.

In the second step, the PV homs are processed in parallel using the multiple SIMD mode. In this step, each PV hom’s keywords are matched against 100 to 200 grammatical patterns per environment, producing scores that evaluate the quality of the match between the PV hom and the grammatical patterns. At the end of this step, all of the PV hom/grammatical pattern pairs such as “add age two year/grammatical pattern 3” and “’add age to year’/grammatical pattern 4” have been generated and scored (See also Figure 6). The PV hom/grammatical pattern pair’s scores are thresholded to reduce the number of pairs. After pruning, the result is about five or six (say $n$) PV hom/grammatical pattern pairs for each of the $m$ PV homs or 15 to 24 ($m \times n$) total PV hom/grammatical pattern pairs (the search tree at this point could be as large as several hundred PV hom/grammatical pattern pairs if the tree were not pruned to eliminate unlikely pairings).

During the third step, each of the $m \times n$ hom/grammatical pattern pairs is processed using MIMD parallelism to generate about three or four (say $o$) different possible parsings of the input sequence into PV and TRCC segments. For example, assume that Plain English has input the associative calculus table shown in Figure 6. (The “Add in costs” and “To be deducted” labels in the “Taxes” table in Figure 5 are somewhat ad hoc, but they illustrate that there is no restriction on PV keywords appearing in column or row names. These names are included in the

Figure 4. INP MIMD/SIMD Parallelism.
table to demonstrate one way that multiple parsings of the same PV hom sequence may occur.

The first terminal symbol in the grammatical pattern in Figure 6 (ADD) matches the first word in the PV hom exactly, but the second symbol in the grammatical pattern (arg1) does not match the second word in the PV hom (ADD). This signals the end of the first PV segment and the beginning of the first TRCC (arg1) segment. Parsing continues in this manner until the end of the sequence.

<table>
<thead>
<tr>
<th>Taxes</th>
<th>Income</th>
<th>Add in costs</th>
<th>To be deducted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joan</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5 – The Taxes Table

The “ADD” grammatical pattern:

ADD arg1 TO arg2 dot

A PV hom

ADD ADD IN COSTS TO TO BE DEDUCTED .

Figure 6 – A PV hom/grammatical pattern Pair

A frame data structure records the “break points” of the parsing, that is, where in the PV hom the segments begin and end, but the segments are not extracted during this step. If multiple parsings are possible for the PV hom, as would be the case for the duplicate “TO” in “ADD ADD IN COSTS TO TO BE DEDUCTED .”, the multiple frame partitionings (See Figure 7) are stored in a frame vector. On average this step results in 45 to 96 \((m \times n \times o)\) different PV hom/grammatical pattern/alternative frame triples to be scored.

In the fourth step, each of the parallel parsing tasks extract the PV and TRCC segments from the PV hom. There are on average 2 \((say \ p)\) TRCCs (e.g. arg1 and arg2 in Figure 6) per PV hom for a total of 90 to 192 \((m \times n \times o \times p)\) TRCCs.

Fifth, the extracted TRCC segments are homonymized to generate the TRCC homs – about 1.5 \((say \ q)\) homs per TRCC segment on average.

In the sixth step, all 135 to 288 \((m \times n \times o \times p \times q)\) TRCC homs are compared to about 200 entries per lexicon using SIMD parallelism. The 135 to 288 associative parallel tasks are equivalent to about 27,000 to 57,600 MIMD tasks. Initially, the scores of the comparisons are stored with the matched lexicon entries. After all lexicon comparisons have been made, the best match for each TRCC is selected and its lexicon relevant information such as the TRCC address and score is moved to the appropriate frame slot.

\[
\begin{array}{|c|c|c|c|}
\hline
VP1 & TRCC1 & VP2 & TRCC2 \\
\hline
ADD & ADD IN COSTS & TO & TO BE DEDUCTE D \\
\hline
ADD & ADD IN COSTS TO & TO & BE DEDUCTE D \\
\hline
\end{array}
\]

Figure 7 – Possible Multiple Partitionings of a PV hom

Seventh, after all PV hom/grammatical pattern/frame/lexicon combination scores have been recorded, the best PV hom/grammatical pattern/frame/lexicon combination is determined by adding the TRCC slot scores to the PV segment scores for each frame and selecting the combination with the lowest (best) frame score.

The last step in the INP algorithm is a function of the grammar in the selected best PV hom/grammatical pattern/frame/lexicon combination and the associated environment databases. The best frame is translated into a verbal or textual form (or both) and echoed back to the user to verify that the input was understood correctly. Upon verification, further action is a function of the understood meaning of the input sequence and is determined by the grammatical pattern. If the grammatical pattern calls for an action, the appropriate action is taken. For example, if it is a search action, the databases are searched for the requested information. If it is a command action, the command for the appropriate output device is generated, etc.

This brief description describes, in our experience, the average range of parallel tasks required by the INP algorithm in a heterogeneous multi-core environment. In contrast, using MIMD or SPMD parallelism would result in about 30,000 to 40,000 parallel threads (SIMD data parallelism with \(y\) objects is equivalent to \(y\) MIMD or SPMD parallel threads). These numbers are estimates of the average fan outs and are intended to illustrate the extent of the parallelism. To bring this number under control, the algorithm allows pruning and/or ranking at each step. Ranking allows the algorithm to be executed on conventional architectures (as is reported here). Pruning allows multiple parallel paths to be explored on heterogeneous multi-core chips because the alternatives can be reliably reduced to about 1800 separate combinations.

5 Results

The INP algorithm was developed and simulated on a conventional computer using a conventional voice recognition system. Table 1 shows a typical set of sample voice inputs to the voice recognition software and the corresponding actual misrecognized ASR outputs that are the INP input sequences. During
debugging, the misrecognized sequences were often input from a keyboard or from a data file for expediency and consistency. It is difficult to get a command to be misrecognized in the same way over and over again. Table 2 gives the results and the overall average for 9 tests. The tests consisted of the same sequence of commands all input orally using a standard voice recognition system. There were 33 commands with 236 words per test. The commands are listed in Appendix A.

### 6 Conclusions and Future Research

INP is a unique algorithm designed for heterogeneous multi-core chips. It is an integration of traditional SIMD and MIMD tasks for post processing ASR output in a high level pseudo natural language programming environment. By counting the number of tasks noted in Figure 4, we estimated that the heterogeneous approach reduces the number of parallel tasks by a factor of about 150 from about 30,000 to 60,000 tasks to 200 to 400 tasks through the use of data parallelism.

The algorithm was developed and simulated on a conventional computer using a conventional voice recognition system, as a result it can be slow when processing input sequences with 3 or more errors. However, it achieved over 99% correct word recognition and over 94% “error” free command recognition. These rates are adequate for personal use when programming PCs, using PDAs, cell phones, electronic books, and other digital devices. However, in critical applications such as performing financial tasks or in military situations, a verification mode is needed. In this mode, after an input has been processed, it is echoed, audibly, visually or both, back to the user for verification.

In general, and especially in critical situations, it is desirable to achieve results closer to 99% correct command recognition. The next research steps are to 1) implement INP using a heterogeneous multi-core chip, such as the Cell chip to achieve actual real time execution, 2) extend INP to web and operating system commands, and 3) identify and address those items that are “problem” areas. Using the sequential implementation, some “noisy” inputs can take more than a minute to process. It is assumed that a multi-core chip implementation will reduce this time to acceptable levels. Also, it is assumed that larger and more complex databases can be accommodated simply by increasing the sizes of the MIMD and/or SIMD components of the heterogeneous environment. Determining the parameters involved and their ranges is an important next step for achieving a useful paradigm.

<table>
<thead>
<tr>
<th>Tests</th>
<th>Percent Correct without INP</th>
<th>Percent Correct with INP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>87.30% 27.30%</td>
<td>100.00% 100.00%</td>
</tr>
<tr>
<td>Test 2</td>
<td>85.20% 24.20%</td>
<td>99.20% 93.90%</td>
</tr>
<tr>
<td>Test 3</td>
<td>85.60% 30.30%</td>
<td>99.60% 97.00%</td>
</tr>
<tr>
<td>Test 4</td>
<td>83.90% 9.10%</td>
<td>99.20% 93.90%</td>
</tr>
<tr>
<td>Test 5</td>
<td>86.00% 18.20%</td>
<td>98.70% 90.90%</td>
</tr>
<tr>
<td>Test 6</td>
<td>87.70% 21.20%</td>
<td>98.30% 87.90%</td>
</tr>
<tr>
<td>Test 7</td>
<td>88.60% 30.30%</td>
<td>99.60% 97.00%</td>
</tr>
<tr>
<td>Test 8</td>
<td>87.70% 24.20%</td>
<td>99.20% 93.90%</td>
</tr>
<tr>
<td>Test 9</td>
<td>87.70% 21.20%</td>
<td>99.60% 97.00%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>86.60% 22.90%</strong></td>
<td><strong>99.20% 94.60%</strong></td>
</tr>
</tbody>
</table>

### Table 1 – Sample Inputs

<table>
<thead>
<tr>
<th>SAMPLE VOICE INPUTS</th>
<th>MISRECOGNIZED VOICE RECOGNITION OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add jane age to size .</td>
<td>Add jane h. or size .</td>
</tr>
<tr>
<td>set john age to 2 .</td>
<td>set john agent to 2 .</td>
</tr>
<tr>
<td>add test_105 jane add in to size .</td>
<td>add and in jane aged to size .</td>
</tr>
<tr>
<td>add jane age to 5 .</td>
<td>that jane h. to five .</td>
</tr>
<tr>
<td>subtract age from john size .</td>
<td>subtract h. a john size .</td>
</tr>
<tr>
<td>set age to 245 .</td>
<td>set age two to 45 .</td>
</tr>
<tr>
<td>label row 2 with general electric .</td>
<td>label role to live general electric .</td>
</tr>
<tr>
<td>label column two with age .</td>
<td>label: 2 with age .</td>
</tr>
</tbody>
</table>

### 7 References


Appendix A

label row 3 with value line asset allocation
label row 4 with price waterhouse
label row 5 with fidelity puritan
label row 6 with total assets
label column 2 with number of shares
label column 3 with price per share
label column 4 with value
label column 5 with cost
label column 6 with net profit

plain english

set general electric number of shares to 100
set general electric price per share to 29.11
set general electric cost to 27.56
set value line asset allocation number of shares to 543.21
set value line asset allocation price per share to 15.33
set value line asset allocation cost to 16.25
set price waterhouse number of shares to 250
set price waterhouse price per share to 25.00
set price waterhouse cost to 24.75
set fidelity puritan number of shares to 200
set fidelity puritan price per share to 15.91
set fidelity puritan cost to 15.51

multiply number of shares by price per share
put the result in value
sum value
put it in total assets value
subtract cost from price per share
multiply by number of shares
put the result in net profit
sum net profit
put it in total assets net profit

goodbye