データストリーム処理系を用いたスケーラブルな音声処理

西井 俊介† 鈴村 豊太郎†‡

†東京工業大学 〒152-8550 東京都目黒区大岡山 2-12-1
‡IBM 東京基礎研究所 〒242-8502 神奈川県大和市下鶴間 1623-14
E-mail: †nishii.s.aa@m.titech.ac.jp, ‡suzumura@cs.titech.ac.jp

あらまし 本研究では，時々刻々と流れるデータに対してリアルタイムに計算処理などの操作を行う事が出来るデータストリーム処理の処理系を用いて並列音声認識システムを実装した．処理系の記述力によって音声認識の並列分散，処理の拡張が容易に行えることを示し，認識処理を4ノード16コアから構成される分散並列環境上でシングルコアと比較し、13.8倍のスループットが得られた．また，データストリーム処理のスループットを大きく保ちつつ，認識精度を過度に低下させすぎないようにするために，音声認識のビームサーチにおけるビーム幅に着目し，入力データレートが大きいときにはビーム幅を下げることによって，認識精度を犠牲にスループットを向上させる機構を実装した．この機構によりスループットと認識精度の両方を保つことが出来ることを実験的に示した．

キーワード データストリーム処理，DSMS，DSPS，System S，SPADE，音声認識，ビームサーチ，Julius

Highly Scalable Speech Processing on Data Stream Processing System

Shunsuke NISHII† and Toyotaro SUZUMURA†‡

†Tokyo Institute of Technology 2-12-1 Ookayama, Meguro-ku, Tokyo, 152-8550 Japan
‡IBM Research - Tokyo 1623-14 Shimotsuruma, Yamato-shi, Kanagawa, 242-8502 Japan
E-mail: †nishii.s.aa@m.titech.ac.jp, ‡suzumura@cs.titech.ac.jp

Abstract In this paper we describe the implementation and evaluation of a Julius-backended parallel and scalable speech recognition system on the data stream management system ``System S'' developed by IBM Research. Our experimental result on our parallel and distributed environment with 4 nodes and 16 cores shows that the throughput can be significantly increased by a factor of 13.8 when compared with that on a single core. We also demonstrate that the beam management module in our system can keep throughput and recognition accuracy with varying input data rate.

Keyword Data Stream Processing, DSMS, DSPS, System S, SPADE, Speech Recognition, Beam Search, Julius

1. 研究の背景

今日，ありとあらゆる場所で大規模同時的に電子的な音声のやり取りが行われており，そのような音声のいくつかに何らかの音声処理を施す技術が求められている．例えば，企業のコールセンターにおける顧客とオペレータの対話に対して音声認識，テキスト処理を加えることでオペレータが伝えるべき内容を伝えているか，コンプライアンス違反となるような発言をしていないかを確認したり，対話中のキーワードからオペレータが必要とする情報をオペレータのモニタ上に表示するなど，コールセンターの品質向上のための様々な用途に用いることができる．このように，リアルタイムの音声処理の有用性は高いといえる．これに対し，データストリーム処理[1]と呼ばれる，時々刻々と流れる大量のデータをストレージに保管することなくオンメモリ上でリアルタイムに処理する計算パラダイムが昨今活発に研究されており，IBM の System S[2][3]や M.I.T.の Borealis[1]などのソフトウェア処理系およびプログラミングモデルが提案されている．

従来もリアルタイム音声処理の研究はなされてきた[4]が，このような音声処理を，データストリーム処理のミドルウェアを用いれば様々な処理を統括的に，高い拡張性を有しながら管理することができる．そこで，本研究ではデータストリーム処理系 System S を基に，汎用的な音声認識オペレータを実装し，スケーラビリティの評価のため簡単な並列音声認識システムを実装した．

また通常，音声認識サーバの性能指標（認識精度，スループット，遅延が発生していない時の応答時間）は同一のシステムの立ち上げ以降は固定である．ここで認識精度とスループットはトレードオフの関係にあるため，スループットを高くするには認識精度を犠牲にする必要が生じる．入力データレートが一定することが分かっている場合は問題はないのだが，入力データレートが状況によって変化する場合，入力データレートが低いときはスループットを高くする必要はなくなる．一方，入力データレートが高いときに認識精度を高く，スループットを低くしていると，処理に遅延が生じ，応答時間が増大してしまう．そこで，このような環境では状況に応じて認識精度とスループットの優先順位を変化させる機構が有効である．そこで，本研究では認識器の認識精度と処理時間に最も影響を与えるパラメータである“ビーム幅”に着目し，現在の入力データレートに応じて最適なビーム幅を設定する機構を実装した．

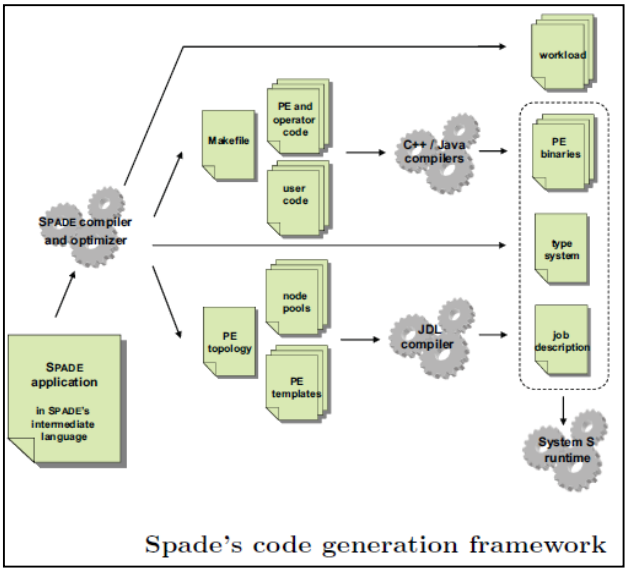

図1　SPADEコンパイラ（[6]より引用）

以降の章では，まず第2章で関連研究について述べる．第3章ではデータストリーム処理について述べ，第4章では音声認識について簡単に説明する．第5章では設計と実装，第6章では評価，第7章では本論文のまとめを述べる．

2. 関連研究

[4]は音声処理のスケーラビリティに関する研究であり，サーバの処理能力とスケーラビリティ評価モデルから，想定するサービスにおいて必要となるサーバの台数を求める手法を提案している．本研究では，データストリーム処理を汎用的に扱うことのできるデータストリーム処理のミドルウェアを用いた点，およびサーバ1台当たりの処理能力を状況に応じて変化させている点で当研究と異なる．

[5]など，データストリーム処理系におけるロードシェディングに関する研究は多くなされている．ロードシェディングでは，入力データレート増大時に流れてくるデータの一部を削除して，システムの遅延を防ぎリアルタイム性を守ることをしている．本研究におけるビーム幅管理モジュールもまた異なる方法ではあるが，入力データレート増大時にリアルタイム性を守るという点でロードシェディングと類似しているといえる．

3. データストリーム処理と System S

3.1 データストリーム処理

データストリーム処理[1][2][3][6][7]とは，止め処なく生成される情報の流れをストリームと呼び，このストリームを蓄積することなく逐次処理していくという新しい計算パラダイムである．バッチ処理と呼ばれる計算対象を全てストレージに蓄積してから計算する従来の手法と違い，リアルタイムの応答が要求される場合や，時系列で前後する僅かなデータの

```
  [Nodepools]
  nodepool Computer[] := ("node0", "node1")

  [Application]
  stream TCPData(data0 : Integer, data1 : Float)
:= Source()["ctcp://some_address", csvFormat]{}
  -> node(Computer, 0), partition["part0"]

  stream LimmitedData(data0 : Integer)
:= Functor(TCPData)[data1 >= 1.5]{}
  -> node(Computer, 0), partition["part0"]

  stream Average(avg : Integer)
   := Aggregate(LimmitedData<count(10),count(1)>)
                              []{avg := Avg(data0)}
    -> node(Computer, 0), partition["part1"]

  Nil := Sink(Average)["file:///average.csv",
csvFormat]{}
  -> node(Computer, 1), partition["part2"]
```

図2　SPADEプログラムの一例

みを参照すればよい計算や，全データの蓄積が物理的に困難な処理に適している．このような手法は音声や動画のストリーミングなど一部の処理では利用されていたが，データストリーム処理はこれを抽象・汎用化し，幅広い処理に対して適用できるよう洗練された処理系としてまとめられている点が従来とは異なっている．

このような処理系を DSMS / DSPS (Data Stream Management / Processing System)と呼ぶが，その一例として M.I.T.の Borealis[1]や IBM Research の System S[2][3]などが存在し，ここ数年活発な研究がなされている．多くの DSMS がシングルノードでの実行を前提としているが，Borealis と System S は分散環境上で実行可能である．

3.2 System S と SPADE

System S[2][3]は，データフロー図から直感的に処理を記述できる SPADE[6]という言語と，自動性能最適化機構を持つ SPADE コンパイラ，処理基盤である SPC[7]によって構成される．

SPADE は高級な宣言的言語で，処理対象であるストリームと，処理を行うオペレータの関係を記述するだけでデータストリーム処理を実行でき，ノード間やプロセス間の通信や，デーモンの立ち上げなどを意識することなくプログラミングが可能である．広範な処理に適用可能な汎用の組み込みオペレータを持つため，単純な処理ならば組み込みオペレータにパラメータを設定するだけで実装できる．汎用オペレータだけでは不十分な場合は，C++や Java を用いたユーザ定義の独自のオペレータや関数の作成もサポートされている．SPADE では，図1のようにコンパイルや最適化を段階的に行うことで高度な最適化を施す．SODA[2]では実行中のノード割り当ての変更のような動的な最適化もサポートしており，処理全体の高速化が図られている．

図2は SPADE プログラムの例で，ある TCP ソケットから

データを受信し, data1 が 1.5 以上のストリームの平均値を求めてファイルに書き出すという処理を表している. [Nodepool]は処理を行うノードを指定する部分で, ここでは二つのノード, node0 と node1 が指定されている. [Application]以降は実際の処理を記述する部分で, stream 文の次に記されているのがストリームの名前とデータタイプで, 次の行がオペレータと処理内容を表している. Source はデータを受け付けるオペレータで, ここではクライアントとしてアドレス some_address へ TCP 接続を行い, CSV 形式のデータを受け取って data0, data1 というストリームを作成する. Source は TCP 以外にもファイルや DB へのアクセスが可能である. その次の行の node や partition はオペレータをどのノードやスレッドで実行するかを指定する部分で, ここでは Computer の 0 番目, つまり node0 で実行することを指定してある. 次のオペレータでも同じ partition を指定しているが, partition が同じオペレータは同一スレッドで動作し, これを System S では FUSION と呼ぶ. 次のオペレータ, Functor はストリーム中のデータに様々な処理を行うオペレータで, ここでは条件文を用いてストリームの選別を行っている. Functor 内ではこれ以外にも複雑な関数を利用でき, 関数そのものもユーザ自身で定義することができる. Aggregate はストリームを一定の区間で区切って, 区切りの中のデータに対して操作を行うオペレータで, ここでは十個ずつのバッファを取ってデータを一つずつ動かしていき, その中での平均値をデータ avg として出力している. このような操作は Windowing と呼ばれる. 最後の Sink オペレータは Source オペレータと対の存在で, データを様々な形式で出力するオペレータである. System S は, 他にもストリームを条件に合わせて複数に分割する Split や, 逆に二つのストリームを一つにする Join, ストリームの同期を行う Barrier などの多様な組み込みオペレータとユーザ定義オペレータ UDOP を持ち, 複雑なデータストリーム処理に対応することが可能となっている. UDOP は実際の処理以外の通信やストリームの管理部分が書かれたスケルトンコードを自動生成するので, ユーザは処理部分のみを C++や Java で記述すればよい. これにより, 汎用オペレータでは表現できない複雑な処理や操作を実現でき, UDOP 自体も他のオペレータと同様にモジュール化されるため高度な柔軟性, 再利用性の恩恵を受けることができる.

多くの処理が組み込みオペレータとして用意されており, ノード間やプロセス間の通信をミドルウェアが行うため開発時間を短縮できる点や, ユーザ定義のオペレータや関数によって複雑で柔軟な処理が可能である点, ノードやスレッドの指定が容易である点から System S はデータストリーム処理やミドルウェアの研究に適していると言え, 本研究ではデータストリーム処理系として System S を使用することとした.

4. 音声認識の処理性能と要因

4.1 音声認識の略説

音声認識とは, 人の発話音声を入力としてその発話内容を文章として出力する処理である. 音声認識では, 音声を特徴量ベクトルの時系列 $X = x_1, x_2, \ldots, x_T$ に変換し, 文章を W と置いたときの確率 P(W|X) を最大化するような文章$\tilde{W}$ を求める.

$$\tilde{W} = \mathrm{argmax}_W P(W|X)$$

ここで, P(W|X)を直接求めるのは困難であるため, ベイズ則に基づき次式のように変形する.

$$\tilde{W} = \mathrm{argmax}_W P(X|W)P(W)$$

式中の P(X|W), P(W) を求めるために使用する統計モデルをそれぞれ音響モデル, 言語モデルといい, それぞれ隠れマルコフモデル (HMM; Hidden Markov Model), N-gram モデルを用いるのが主流である.

音声認識エンジンとしては, HTK[8], Julius[9][10], T^3 decoder[11] などが存在するが, 本研究では SPADE 上で UDOP として音声認識オペレータを実装する際の便宜から Julius 4.1.4 を採用した.

4.2 ビームサーチ

Julius では, 音響モデルと言語モデルをもとに探索ネットワークを構築し, 入力音声に対して tree-trellis 探索[12]に基づいたアルゴリズムにより音声認識を行う. アルゴリズムの全体は 2 パスで構成されており, 第一パスでは強い近似などを用いた粗い探索により正解文章の候補を絞り込み, その結果に基づいて第二パスで詳細な認識を行い, 最終的な出力を得る. 第一パス, 第二パスともに, ネットワーク上の解探索時に全ての経路について探索を行うのではなく, スコアの高い仮説のみ計算を行いそれ以外の仮説を棄却する手法をとっている. この探索手法をビームサーチと呼び, ビームサーチにおける足切り幅をビーム幅と呼ぶ.

音声認識処理においてその処理時間の大部分を占めるのが第一パスの処理である. 精度の高い音声認識を行うには, 広い探索空間から正解文が認識仮説の候補として残る確率を高くするために, 第一パスのビーム幅の値を十分に大きくとらなければならない. 一方この値を大きくしすぎると認識に要する時間が大きくなってしまう. そのため, 第一パスにおけるビーム幅の設定は Julius の認識精度および処理時間に最も大きく影響する.

以降の節では, 第一パスにおけるビーム幅のことを単純にビーム幅と呼ぶことにする.

5. 設計と実装

System S および SPADE を用いてスケーラブルな音声認識システムを実装した. 図 3 にシステムの大まかなデータフロー図を示し, 図 4 では個々のオペレータのレベルでシステム

の詳細な構成を示す．また，システムの SPADE ソースコードを Appendix に記載した．Appendix のソースコード中で番号の振られている箇所は，図 4 の対応する番号のオペレータの記述を表わしている．

以下の節では，5.1~5.3 節でシステム設計の理念・議論等を述べ，5.4 節でシステムの各モジュールについて説明する．5.5 節ではさらに掘り下げて，個々のオペレータの処理について説明する．

5.1 拡張性とスケーラビリティ

並列分散処理のスケールアウト性，様々な音声処理への拡張性を考慮してシステムの設計を行った．このシステムの SPADE プログラムの行数はわずか 120 行程度であり，SPADE を用いれば複数台のノードにわたる並列分散処理も簡単に記述できることを示している．また，並列分散処理のスケールアウトを行うには，1~2 行目の並列数の記述および 24~27 行目のノードの記述を変更するだけでよく，このことからも容易にスケールアウトすることができることが分かる．さらに，このシステムは音声認識のみを行うシステムとして実装されているが，ここで 71~75 行目に記述されている (4) “Decoder@j_Transcription” ストリーム (データフロー図の “UDOP SpeechDecoder”) に続く形で処理を追記すれば，音声認識を基にする様々な音声処理システムに拡張することができる．

5.2 音声認識オペレータ

図 3 中の認識器モジュールにおいて音声認識を実行するためには，システムの内部または外部に音声認識を実際に行うプログラムを用意する必要がある．本研究では，3.2 節で述べた UDOP により音声認識オペレータを実装した．このオペレータは，図 4 の(4)SpeechDecoder に該当する．この音声認識オペレータは，本研究におけるシステム以外でも用いることのできるように，汎用的なオペレータとして実装されている．オペレータのバックエンドエンジンとして Julius のライブラリである libjulius, libsent を用いた．また，このオペレータは 5.3 節で後述する理由により，Julius のビーム幅の値をシステム実行中に変更することができるように実装されている．なお，オペレータの実装の際，実装および評価の容易性のため，音声の発話区間 1 つをまとめて入力として受け取り，その区間ごとに入力が完了してから認識処理を開始する方法をとった．

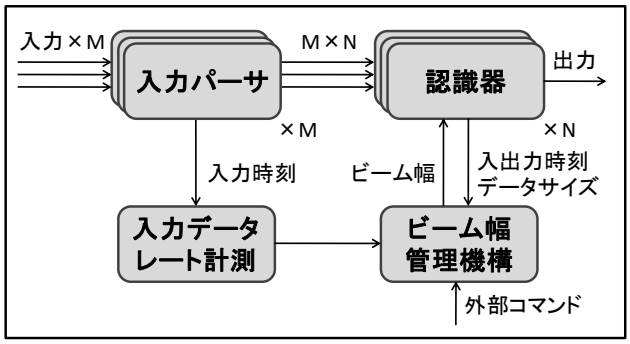

図3　システムの構成

5.3 ビーム幅管理機構

4.2 節で述べた通り，音声認識で最も処理時間を要するのが第一パスの処理であり，その処理時間はビーム幅の設定に依存する．通常，Julius ではビーム幅はユーザによって予め適切な値を設定しておき，エンジンの起動以降同じ値を続けて使用することになる．ここで，ビーム幅の設定によって認識精度，応答時間，スループットが変化するのだが，入力データレートが小さいときは，スループットはあまり大きな値を必要としない．一方，入力データレートが大きいときはスループットを大きくしないと処理が追いつかなくなり，応答時間が増大してしまう．そこで本研究では，5.2 節で述べた

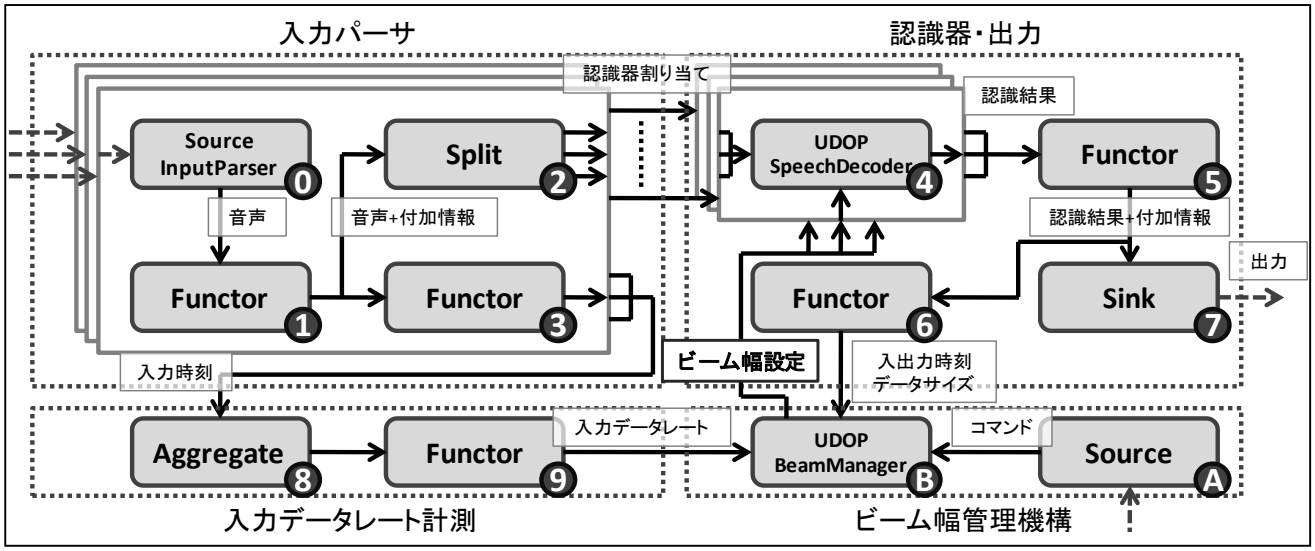

図4　システムの詳細な構成

通り, システムの実行中にビーム幅を再設定できるように音声認識オペレータを実装した. そして, 入力データレートに応じて最適なビーム幅を設定する機構を実装した. この機構もまた音声認識オペレータと同様UDOPで記述されており, 図4中では(B)BeamManagerに該当する. この機構では, まず認識器のビーム幅に設定する値の候補 (例: 400, 800, 1200) をシステム利用者に設定してもらう. そのビーム幅に対して, システム利用者に予め用意してもらった計測用のデータを用いてスループットを計測する. その後, システムの実運用時には, 入力データレートを監視して, スループットが入力データレートを下回らない範囲で最大のビーム幅を選択し, その値が現在の認識器のビーム幅と異なる場合は認識器に対してビーム幅を再設定する命令を送る.

5.4 システムの構成

図3に示す通り, システムは大きく分けて“入力パーサ”, “認識器”, “入力データレート計測”, “ビーム幅管理”の4つのモジュールで構成されている. 以下ではシステムの各モジュールについて説明する.

まず, 入力パーサはソケット通信により入力を受け付け, System S 内部で取り扱えるデータ形式に変換するモジュールであり, M コア（M スレッド）並列に動作する. 入力パーサの各スレッドはそれぞれ独立にソケット通信のポートを持っており, 1つのスレッドにトラフィックが集中することを防ぐことができる. また, 入力パーサでは計算機のタイマーを使って入力時刻をデータに付与する. この入力時刻は入力データレートやスループットの測定に用いられる.

次に, 認識モジュールは5.2節で実装した音声認識オペレータを用いたモジュールであり, N コア（N スレッド）並列に動作する. N 並列の認識モジュールの入力はM 並列の入力パーサから与えられるが, このときに入力が認識モジュールのどのスレッドに割り当てられるかはナイーブなラウンドロビン法に基づき決定する. すなわち, 入力パーサは入力データを処理するごとに認識モジュールの 1 番, 2 番, …, N番とデータを割り当てていき, N番の次は再び1番に戻る, という風に割当先を決める. 認識モジュールは音声認識の完了後, その認識結果を出力する. また, 出力時に計算機のタイマーを使って出力時刻を求め, 入力時刻, 出力時刻および音声の長さをビーム幅管理モジュールに与える. これらの値は, スループットの計測に用いられる

入力データレート測定モジュールは, 入力時刻をもとに現在の入力データレートを測定する. ビーム幅管理モジュールは外部からのコマンド入力により, (1)ビーム幅の設定, (2)現在のビーム幅に対するスループットの計測, (3)現在の入力データレートに対して最適なビーム幅を設定, の3通りの動作をする. このうちの(3)では, (2)で測定したスループットの値から現在の入力データレートに対して処理可能な最大のビーム幅を選択する.

5.5 システムを構成するオペレータ

図 4 はシステム全体の詳細なデータフロー図を示している. システム全体は 12 項目のオペレータにより構成されており, 図では各オペレータに (0)〜(9), (A), (B) と番号を割り振られている. このうち (0)〜(3) は入力パーサ, (4)〜(7) は認識モジュール, (8), (9) は入力データレート計測モジュール, (A), (B) はビーム幅管理モジュールに属する.

各オペレータの動作の説明を以下に記す.

(0)システム外部からの音声入力を System S 内部で取り扱うためのタプル形式に変換.
(1)入力時刻と認識器割り当て用のタグを付加.
(2)音声入力を各認識器に分割.
(3)入力データレート計測用のデータを取り出す.
(4)音声認識を行う. ビーム幅の設定可能.
(5)出力時刻を付加.
(6)スループット計測用のデータを取り出す.
(7)システム外部に認識結果を出力.
(8)入力情報を一定の窓幅で集計.
(9)入力データレート計測.
(A)システム外部からコマンドを入力.
(B)データを集積し, 必要ならばビーム幅を設定.

6. 評価

システムの認識モジュールのスレッド数に対するスループットのスケールアウト性, およびビーム幅管理モジュールの動作について実験的に評価した.

6.1 実験環境

実験時の計算機環境として, システムの外部で入力を行い出力を受け取るノードに Opteron 1.6GHz L2 512KB (2 cores), Memory 8GB を 1 台, 入力パーサの動作, および出力時刻の測定に用いるノードに Phenom X4 2.0GHz L2 512KB (4 cores), Memory 3.5GB を 1 台, 認識モジュールの動作に用いるノードに Phenom X4 2.5GHz L2 512KB (4 cores), Memory 8GB を 4 台, 入力データレート測定およびビーム幅管理モジュールの

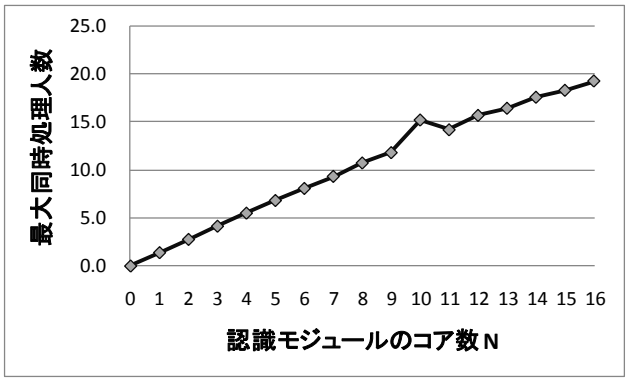

図5 認識モジュールのスケールアウト性

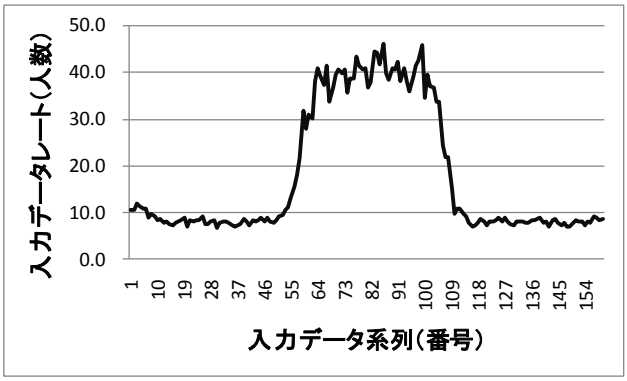

図6　set1の入力データレートの変化

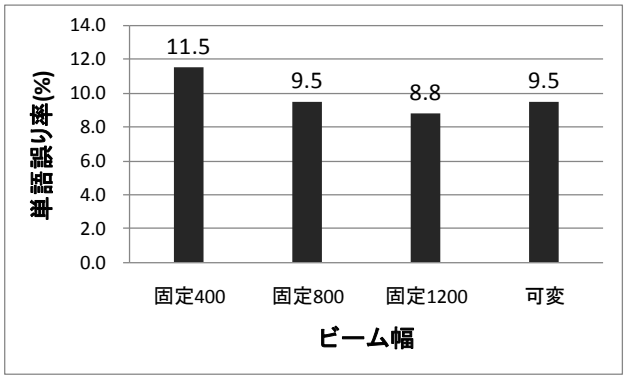

図7　set1の認識誤り率

動作に用いるノードにPhenom X4 2.5GHz L2 512KB (4 cores), Memory 8GBを1台用いた．また，計算機環境全体のネットワーク環境として，1Gbpsのイーサネットで接続されている．ソフトウェア環境としては全ノード共通で，CentOS 5.2 2.6.18-92.el5 AMD64, gcc4.1.2, InfoSphere Streams 1.2 (System S) を用いた．

6.2 認識モデルとパラメータ

音響モデルとして，日本語話し言葉コーパス(JNAS)[13]に収録されている52.4時間（うち，男性話者25.1時間，女性話者 27.3 時間）の音声に基づき学習した，特徴量ベクトル38次元（MFCC+Δ+ΔΔ+ΔE+ΔΔE），16混合ガウス分布，総状態数3000のtriphone HMMを用いた．言語モデルとして，毎日新聞社の1991～2002年の新聞記事データに基づき学習した6万語彙の3-gramを用いた．また，ビーム幅を除くその他のJulius設定パラメータは，ビーム幅の値が1200以下の時に後述する評価用データセットに対してRTFが1.0以下となる値を設定した．

6.3 評価用データセット

評価のためにシステムの入力に与えるデータセットとして，JNASに収録されているIPA-98-TestSetを用いた．これは音響モデルの学習に用いた音声とは重複しない音声である．実験ではこのデータセットをset0，set1の2つに分けて使用した．set0はIPA-98-TestSetのうち約20%の音声（男性1.8分，女性1.9分，合計3.7分）で，set1は残りの約80%の音声（男性7.9分，女性8.6分，合計16.5分）で構成される．

6.4 スケールアウト性の評価

入力パーサの並列数Mを4とし，ビーム幅を1200に固定して，認識モジュールの並列数Nを1～16まで変化させたときの認識モジュール全体のスループットを計測した．入力データとして，入力パーサのどのポートにもset0全体を用いた．図5に結果を示す．縦軸の値はスループットで単位は最大同時処理人数であり，この値はRTFの逆数と一致する．図より，シングルコア動作時の最大同時処理人数は1.4，4ノード16コア並列動作時の最大同時処理人数は19.3であった．これは，16コア並列動作時のスループットはシングルコア動作時の13.8倍となることを示している．なお，この際のset0に対する単語誤り率は5.9%であった．

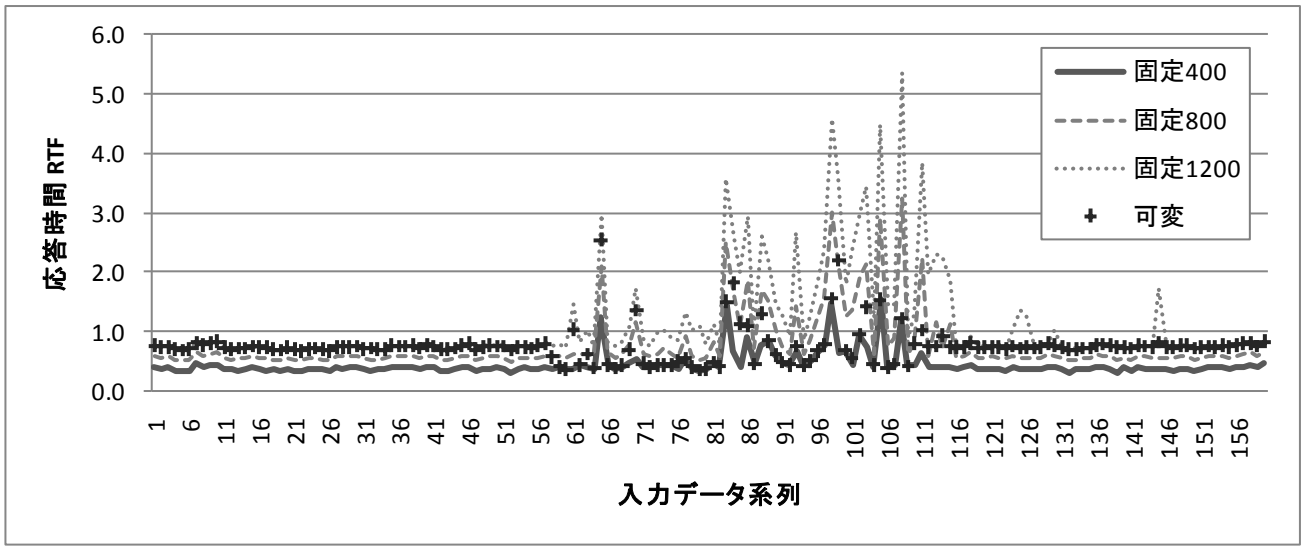

図8　set1に対する応答時間の実時間係数（応答時間÷音声時間）

6.5 ビーム幅管理モジュールの動作の評価

ビーム幅管理モジュールの動作により, 入力データレートに応じてビーム幅を再設定することによって, 高データレート時にも応答時間が増大することなくシステムが動作することを実証した．入力パーサの並列数 M は 4，認識モジュールの並列数 N は 16 とした．設定するビーム幅の候補として，400, 800, 1200 のうち現在の入力データレートに対して最適なものを選ぶようにした. この際にスループットを計測するために set0 を用いた．テストセットとして set1 を用いるが，ここで，入力を与える速度を図 6 に示す通りに変化させて行った. 以上の条件におけるテストセットに対する応答時間および単語認識誤り率を, ビーム幅の値を 400, 800, 1200 の値に固定した場合との比較を行った.

テストセットの各音声に対する単語誤り率を図 7 に, 応答時間を図 8 に示す．なお，この応答時間は，各発話区間における発話終了からシステムの出力までの時間とする. ビーム幅可変の場合，低データレート時にはビーム幅が高く，高データレート時にはビーム幅が低く設定されるので, 図 7 に示す通り，単語誤り率はビーム幅を 400（可変幅の最小値）に固定した場合よりも小さく，ビーム幅を 1200（可変幅の最大値）に固定した場合よりも大きくなった．また，図 8 の結果より，ビーム幅を可変にした場合，1200 固定の場合と比べて, 高データレート時にも応答時間の増大は抑えられている．このように，ビーム幅管理モジュールによりスループットと認識精度の両方を保つことができる．なお，ビーム幅の値をどのように設定した場合でも, 高データレート時にはスパイク的に応答時間の増大が生じている．これは，認識器のタスク割り当てに 5.4 節で述べた通りナイーブなラウンドロビンを用いているのが原因であると考えられる．すなわち，タスク割り当て時に一部の認識器に負荷が集中してしまい，結果としてその認識器に割り当てられた音声の処理に遅延が生じてしまうのだと考えられる.

7. まとめと今後の課題

本研究では，データストリーム処理系 System S に基づいて Julius 4.1.2 をバックエンドとして使い, 汎用的な音声認識オペレータを実装した．また，その音声認識オペレータを用いて並列分散音声認識システムを構成し, そのシステム内にはスループットと認識精度を保つためのビーム幅管理モジュールを実装した．評価実験により，実装したシステムの認識モジュールのスレッド数に対するスケールアウト性は, 16 スレッド並列動作時のスループットがシングルスレッド動作時の 13.8 倍になる程度であることを示した．さらに，実験によりビーム幅管理モジュールがうまく動作しスループットと認識精度の両方を保つことができることを示した.

今回の実装では, 入力パーサから認識モジュールへのタスクの割り当てにナイーブなラウンドロビン法を用いた. このような方法では, 認識モジュールのスレッドごとにタスク量のばらつきが生じてしまい, 応答時間が安定しなくなる問題がある．これを防ぐための方法として，入力データの音声時間をタスク量と近似して, 各スレッドに割り当てられたタスクの合計の音声時間が均等になるように割り当てるという方法が考えられる．他には，各スレッドにおいて処理待ちデータキューの大きさを見て, 最も小さいスレッドにタスクを割り当てる方法などが考えられるが，これは SPADE の仕様上通常の方法では実装できない.

また，今回実装した音声認識オペレータでは，音声入力を一括して受け取ってから認識処理を始める方式を採用した．この方式は，発話開始から逐次入力として受け取り，処理を進める方式と異なり, 発話完了からの応答時間が大きくなってしまうという問題がある．そのため，逐次処理可能な音声認識オペレータ, および音声認識システムを実装する必要がある.

実装にあたって，UDOP のデータ駆動方式と libjulius のコールバック方式の兼ね合いの悪さから，libjulius ベースの逐次処理可能な音声認識オペレータの実装は困難である. そのため, 逐次処理可能な音声認識オペレータをじっそうするためには以下のような方法が考えられる．一つは，自前で音声認識エンジンを実装する，あるいは libjulius を大きく改造する方法であるが，これは既存の資源を有効活用できず，コストのかかる手法である．もう一つは，libjulius ベースの音声認識エンジンを System S の外部で実行できるように実装し，エンジンとシステムのやりとりを UDOP, source オペレータを用いて実装する方法が挙げられる. 後者の方法であれば前者よりも実装が容易であり，また UDOP がエンジンのタスク割り当ての管理を行うことで適切なタスク管理を実装できるという利点もある.

なお，逐次処理方式では認識器 1 スレッドあたりの CPU 使用率があまり大きくならない．このため，逐次処理では一括処理のように 1 コアあたり 1 スレッドで固定するのではなく，より多くのスレッドを立ち上げた方が効率が良くなる．ここで, 計算機の処理限界を超えない範囲で稼働できるスレッド数の最大の値が，認識器全体のスループットとなる.

謝辞

本研究の一部は科学研究費補助金・挑戦的萌芽研究（課題番号:22650017) の助成によって行われた.

参 考 文 献

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. CIDR*, pages 277–289, 2005.

[2] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu and L. Fleischer. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. *Middleware*, 2008.

[3] B. Gedik, H. Andrade and K.-L. Wu. A Code Generation Approach to Optimizing High-Performance Distributed Data Stream Processing. In *Proc. USENIX*, pages 847-856, 2009.

[4] 荒金 陽助, 下川 清志, 金井 敦. 音声対話システムにおけるスケーラビリティ評価モデルの検討. 情報処理学会論文誌 46(9), pp. 2269-2278, 2005.

[5] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. VLDB*, 2003.

[6] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu and M. Doo. SPADE: The System S Declarative Stream Processing Engine. In *Proc. SIGMOD*, pages 1123-1134, 2008.

[7] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park and C. Venkatramani. SPC: A Distributed, Scalable Platform for Data Mining. *DM-SSP*, pages 27-37, 2006.

[8] S. Young, G. Evermann, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev and P. Woodland. The HTK book (for HTK Version 3.2). 2002.

[9] A. Lee and T. Kawahara. Recent Development of Open-Source Speech Recognition Engine Julius. *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2009.

[10] 李 晃伸. 大語彙連続音声認識エンジン Julius ver.4. 電子情報通信学会技術研究報告. SP, 音声 107(406), pp.307-312, 2007.

[11] P. R. Dixon, D.A Caseiro, T. Oonishi, S. Furui. The Titech Large Vocabulary WFST Speech Recognition System. *IEEE ASRU*, pages 443-448, 2007.

[12] A. Lee, T. Kawahara and S. Doshita. An Efficient Two-pass Search Algorithm using Word Trellis Index. In *Proc. ICSLP*, pages 1831-1834, 1998.

[13] 板橋 秀一, 山本 幹雄, 竹沢 寿幸, 小林 哲則. 日本音響学会新聞記事読み上げ音声コーパスの構築. 日本音響学会研究発表会講演論文集 1997(2), pp. 187-188, 1997.

Appendix

システムの SPADE ソースコード

```
1  #define SRC_NUM 4
2  #define DEC_NUM 16
3
4  #define INPUTPARSER_BLOCKSIZE 512*1024
5  #define DEFAULT_BEAM 400
6  #define INPUTRATE_AGG_RANGE 10
7
8  #define SRC_HOST st00
9  #define CMD_HOST se00
10 #define SNK_HOST sa07
11
12 [Application]
13 ssr
14
15 [Libdefs]
16 incpath "~/udoplib/julius-4.1.4/libjulius/include/"
17 incpath "~/udoplib/julius-4.1.4/libsent/include.fix/"
18 libpath "~/udoplib/julius-4.1.4/libjulius/"
19 libpath "~/udoplib/julius-4.1.4/libsent/"
20 libs "julius"
21 libs "sent"
22
23 [Nodepools]
24 nodepool n_src[] := ("st00")
25 nodepool n_dec[] :=
26 ("st05","st06","st07","st08","st05","st06","st07","st08",
27  "st05","st06","st07","st08","st05","st06","st07","st08")
28 nodepool n_misc[] := ("se00")
29
30 [Program]
31
32 # --- INPUT PARSER MODULE (@i) : Input@i*
33
34 for_begin @i 0 to SRC_NUM - 1
35   stream Input@i_Source(id: String, speech: ShortList, nsamples: Integer)
36     := Source() ["stcp://SRC_HOST:627@i/",
37         udfbinformat="InputParser", blockSize = INPUTPARSER_BLOCKSIZE, noDelays] {}
38     -> node(n_src, 0), partition["p_src_r@i"]
39
40   stream Input@i_SpeechTag
41   (id: String, speech: ShortList, nsamples: Integer, itime: Long, selector: Integer)
42     := Functor(Input@i_Source) [] { itime := timeMicroseconds(),
43         selector := mod(toInteger(seqNum()) + @i * DEC_NUM / SRC_NUM, DEC_NUM) }
44     -> node(n_src, 0), partition["p_src_r@i"]
45
46   for_begin @j 0 to DEC_NUM - 1
47     stream Input@i_SplitTo@j
48     (id: String, speech: ShortList, nsamples: Integer, itime: Long, selector: Integer)
49   for_end
50     := Split(Input@i_SpeechTag) [selector] {}
51     -> node(n_src, 0), partition["p_src_r@i"]
52
53   stream Input@i_ITime(nsamples: Integer, itime: Long)
54     := Functor(Input@i_SpeechTag) [] {}
55     -> node(n_src, 0), partition["p_src_r@i"]
56 for_end
57
58 bundle B_Input_ITime := ()
59 for_begin @i 0 to SRC_NUM - 1
60   B_Input_ITime += Input@i_ITime
61 for_end
62
63 # --- SPEECH DECODER MODULE (@j) : Decoder@j*
64
65 for_begin @j 0 to DEC_NUM - 1
66   bundle B_Decoder@j_SpeechTag := ()
67   for_begin @i 0 to SRC_NUM - 1
68     B_Decoder@j_SpeechTag += Input@i_SplitTo@j
69   for_end
70
71   stream Decoder@j_Transcription(id: String, transcription: String,
72   nsamples: Integer, beam: Integer, itime: Long)
73     := Udop(B_Decoder@j_SpeechTag[:] ; BeamMan_Beam) ["SpeechDecoder"]
74     {beam="DEFAULT_BEAM"} -> node(n_dec, @j), partition["p_dec@j"]
75 for_end
76
77 # --- (SINK) : Sink*
78
79 bundle B_Sink_Transcription := ()
80 for_begin @j 0 to DEC_NUM - 1
81   B_Sink_Transcription += Decoder@j_Transcription
82 for_end
83
84 stream Sink_Result(id: String, transcription: String, nsamples: Integer,
85 beam: Integer, itime: Long, otime: Long)
86   := Functor(B_Sink_Transcription[:]) [] {otime := timeMicroseconds()}
87   -> node(n_src, 0), partition["p_src_otime"]
88
89 stream Sink_TrainingData
90 (nsamples: Integer, itime: Long, otime: Long)
91   := Functor(Sink_Result) [] {}
92   -> node(n_src, 0), partition["p_src_otime"]
93
94 Null := Sink(Sink_Result)
95   ["ctcp://SNK_HOST:6281/", csvFormat, noDelays]{}
96   -> node(n_misc, 0), partition["p_misc_sinkt"]
97
98 # --- INPUT RATE MODULE : InputRate*
99
100 stream InputRate_Agg
101 (sum_ns: Integer, last_ns: Integer, max_itime: Long, min_itime: Long)
102   := Aggregate(B_Input_ITime[:] <count(INPUTRATE_AGG_RANGE), count(1)>)
103   [] { Sum(nsamples), Last(nsamples), Max(itime), Min(itime)}
104   -> node(n_misc, 0), partition["p_misc_beam"]
105
106 stream InputRate(irate: Double)
107   := Functor(InputRate_Agg) []
108   { (1000000.0d / 16000.0d) * toDouble(sum_ns - last_ns) / toDouble(max_itime - min_itime) }
109   -> node(n_misc, 0), partition["p_misc_beam"]
110
111 # --- BEAM MANAGER MODULE : BeamMan*
112
113 stream BeamMan_Command(command: String)
114   := Source() ["stcp://CMD_HOST:6280/", csvFormat, noDelays] {}
115   -> node(n_misc, 0), partition["p_misc_beam"]
116
117 stream BeamMan_Beam(beam: Integer)
118   := Udop(BeamMan_Command ; Sink_TrainingData; InputRate)
119   ["BeamSetter"] { beam="DEFAULT_BEAM", agg_range="INPUTRATE_AGG_RANGE" }
120   -> node(n_misc, 0), partition["p_misc_beam"]
```