TR-IT-0023

用例検索の超並列計算機 CM–2 を使った高速化

Acceleration of example-retrieval
on a massively parallel processor, the CM–2

西山 尚哉　　隅田 英一郎
Naoya NISHIYAMA　Eiichiro SUMITA

1993 年 10 月

概要

音声翻訳の実現を目指す上で、大規模な対訳用例を利用することが不可欠であると考えられる。特に、用例検索手法では、用例を用いた自然言語処理 (訳語の選択、文の翻訳、構造的曖昧性の解消等) において有効性が確認されているが、用例数の増加に比例して検索時間が増加するため、対訳用例の高速検索技術を確立しておく必要がある。我々は、細粒度超並列計算機 CM–2 上で対訳用例の高速検索アルゴリズムを提案し、評価実験を試みた。その結果、対訳用例数に関わらず高速な検索が可能となり、大規模用例に対する見通しを得たので、これについて報告する。

エイ・ティ・アール音声翻訳通信研究所
ATR Interpreting Telecommunications Research Laboratories

目次

図目次

表目次

第 1 章

はじめに

用例主導機械翻訳 (EBMT: Example-Based Machine Translation) は、用例に基づいた手法 (EBA: Example-Based Approach) を機械翻訳の訳語選択に適用したものであり、特に訳語選択が難しいとされる「A の B」形式の名詞句の翻訳において、その有効性が確認されている。用例主導機械翻訳は、対訳用例を追加することで翻訳精度を向上させるので、対訳用例の高速検索技術を確立しておくことが重要である。そのために、逐次計算機では探索空間の圧縮によって高速化できても、劇的な性能の向上は望めない。そこで、我々はもう一つの可能性として、並列計算機上での実現を試みた。並列計算機では、複数のプロセッサが相互に通信しながら効率的に処理を進めることができる。特に、CM-2 は SIMD(単一インストラクション / 多重データ) 方式を採用しており、大量の対訳用例データに対して同一の計算を行なう用例検索には、最適のアーキテクチャの一つと考えられる。実際に CM-2 上の高速検索アルゴリズムを提案し、評価実験を行なったので報告する。

以下、第 2章で用例検索の概要、第 3章で CM-2 の概要、第 4 章でプログラム、第 5 章で実験の結果、第 6 章で大規模な対訳用例を使った実験について、第 7 章では、より高速な処理の可能性について考察し、最後に、第 8 章でまとめを行う。

第 2 章

用例検索の概要

用例検索処理は用例を用いた自然言語処理 (特に訳語の選択、文の翻訳、構造的曖昧性の解消等) の基本技術である [2, 4, 5, 6]。本稿では特に「A の B」形式の名詞句の翻訳を題材に用いる。

2.1 訳語選択における用例検索

「A の B」形式の名詞句の訳語選択における用例検索を説明する。

京都 の 会議 → the comference *in* Kyoto
用例検索

1. 入力「京都 の 会議」の「の」の訳し分けをするために、入力に似た (意味的距離が小さい) 用例を検索する。
2. 表 2.1に示した最小距離の用例はすべて「in」で訳されている。
3. システムは、この根拠に従って当該名詞句の訳として、「the comference *in* Kyoto」を出力する。

表 2.1: 「A の B」の翻訳例

意味的距離	日本語	英語
0.4	東京 での 滞在	the stay *in* Tokyo
0.4	香港 での 滞在	the stay *in* Hongkong
0.4	大阪 での 滞在	the stay *in* Osaka

第 3 章

CM–2 の概要

CM–2 はシンキングマシン社が開発した SIMD(単一インストラクション / 多重データ) 方式の超並列計算機である。システムは並列処理ユニット、フロントエンド・コンピュータ (SPARC)、I/O システムから構成されており、フロントエンド・コンピュータ上で実行中のプログラムから、並列動作部分が並列処理ユニットに送られて高速処理される。

CM–2 の特徴は次の通りである。

- 並列処理ユニットには、1 ビットの要素プロセッサを、4K ～ 64K 個の構成で実装することが可能で、システム構成にスケーラビリティがある。また、各要素プロセッサは 1 ビットの ALU(算術論理演算装置) と 8K ～ 32K バイトのメモリ、通信インターフェースなどから構成される。更に、数値演算処理を高速化するために、オプションで単精度 (32 ビット) または倍精度 (64 ビット) の FPU(浮動小数点アクセラレータ) が実装可能である。
- プログラミング言語としては CM FORTRAN、C*、*LISP などが用意されている。
- 通信ネットワークのトポロジは NCUBE、iPSC/2 と同様に N 次元のハイパーキューブ方式 (64K プロセッサ実装時で 12 次元) を採用しており、汎用通信方式の他に直接ハイパーキューブネットワークを用いる高速な通信形態を提供している。

第 4 章

プログラム

我々は超並列連想プロセッサ IXM2、超並列計算機 CM-2 を使って用例検索の高速化を試み、その結果を報告している [7]。この際、IXM2 は高い性能を示したが、CM-2 は性能が低かった。この節では、CM-2 上での用例検索の問題点を明らかにするために、先に報告された CM-2 上での用例検索の問題点を明らかにし、新しい検索アルゴリズムの提案を行なう。

4.1 用例検索のタイプ

用例検索を 3 タイプに分類する。例文中で、括弧内の数字は左の単語の意味概念を表す類語コードであり、下線部分は各タイプにおいて着目している部分である。

- **見出し完全一致検索**
  入力と用例の見出しが一致する。

  入力データ: 京都(004) での 会議(344)
  用例データ: 京都(004) での 会議(344)

- **類語コード完全一致検索**
  入力と用例の類語コードが一致する。

  入力データ: 京都 (004) での 会議 (344)
  用例データ: 香港 (004) での 打合せ (344)

- **類似検索 (上記の 2 つの検索タイプと一致しないもの。)**
  入力と用例の意味距離[1]が最小である。

  入力データ: 京都 (004) での 会議 (344)
  用例データ: 香港 (004) での 滞在 (319)

[1] 意味距離計算の詳細は本稿では扱わない (文献 [5] 参照)。

4.2 用例検索のアルゴリズム

まず用例検索の全体の流れを把握するために、図4.1にアルゴリズムを簡単なブロック図で示す。次に、問題点と改善案を示す。

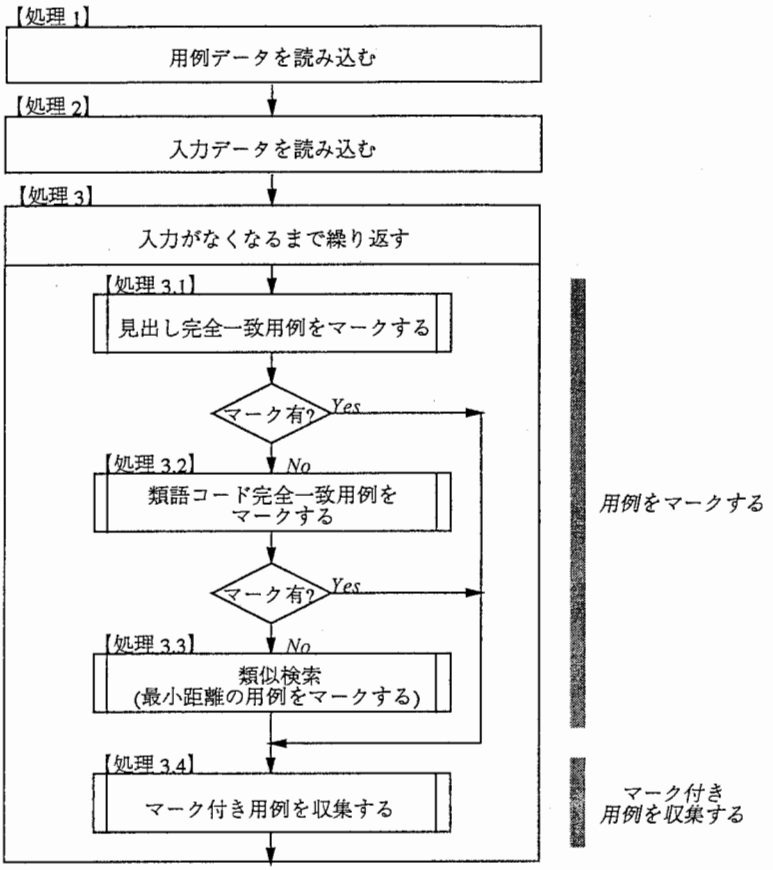

図 4.1: 用例検索のプログラム

【処理 1】 ファイルから用例を読み込み、用例を要素プロセッサに 1 対 1 で割り付ける。

【処理 2】 入力データを読み込み、フロントエンド側のメモリ中に保存しておく。

【処理 3】 以下を入力がなくなるまで繰り返す。

【処理 3.1】 用例が入力の見出しと完全に一致すれば、フラグ変数にマークを付与した後【処理 3.4】へ

【処理 3.2】 用例が入力の類語コードと完全に一致すれば、フラグ変数にマークを付与した後【処理 3.4】へ

【処理 3.3】 それ以外であれば、入力との意味距離を用例ごとに計算しその値を一時変数に保持する。また、意味距離の最小値を求める。

【処理 3.4】 マークされた用例、あるいは最小の意味距離を持つ用例の収集を行なう。

4.3 収集プログラム (改良前)

【処理 3.1】～【処理 3.3】の用例をマーク付与する処理は比較的高速に実行することができる。改良前の【処理 3.4】は、フロントエンド・コンピュータと各要素プロセッサ間の通信を頻繁に使用しており、これが性能を極端に悪くしていた。3つの検索タイプはすべて同様に処理されるので、類似検索を例にとって改良前のプログラムを説明し、次に改良について述べる。

4.3.1 収集プログラム (改良前)

以下に改良前の収集プログラム【処理 3.4】を示す。

【処理 A】 以下をすべての用例に対して繰り返す。

【処理 A.1】 用例が保持している意味距離と最小距離を比較して、等しければデータをホスト側の配列に格納する。

このプログラムの問題は、繰り返し処理や距離の比較を行なう処理がフロントエンド・コンピュータ上でシーケンシャルに実行されるため、用例数に比例して処理速度が増加することである。

4.3.2 収集プログラム (改良後)

前節の問題点を考慮し、改良を行なった収集プログラムを示す。

【処理 B】 処理 B.1 ～処理 B.4 を 1 順に実行する。

【処理 B.1】 要素プロセッサの中で最小値を持つ要素プロセッサを選択し、選択されたプロセッサ数の総計をカウンタに代入しておく。

【処理 B.2】 インデックス配列を用意し、選択されたプロセッサに対応する項目に、プロセッサ番号の昇順に 0 から始まる整数を割り付ける。

【処理 B.3】 プロセッサアドレスのオフセットとして、インデックス配列の値を用いることで、選択されたプロセッサの持つ用例番号を作業用配列の先頭から【処理 B.1】で設定されたカウンタ分をプロセッサ領域に転送する。

【処理 B.4】 作業用配列の先頭からカウンタ分のデータをフロントエンド・コンピュータ側に転送する。

4.3.3 収集プログラム (改良後) のデータの流れ

図 4.2は、改良後の用例収集プログラムでのデータの流れを示している。このアルゴリズムを実現するために、インデックスを格納する変数と用例の再配置のための作業用の変数の 2 変数を追加している。再配置の処理では、収集の対象となる用例を持つ各要素プロセッサが、インデックス変数で指定された要素プロセッサに対して並列にデータ転送を行なう。

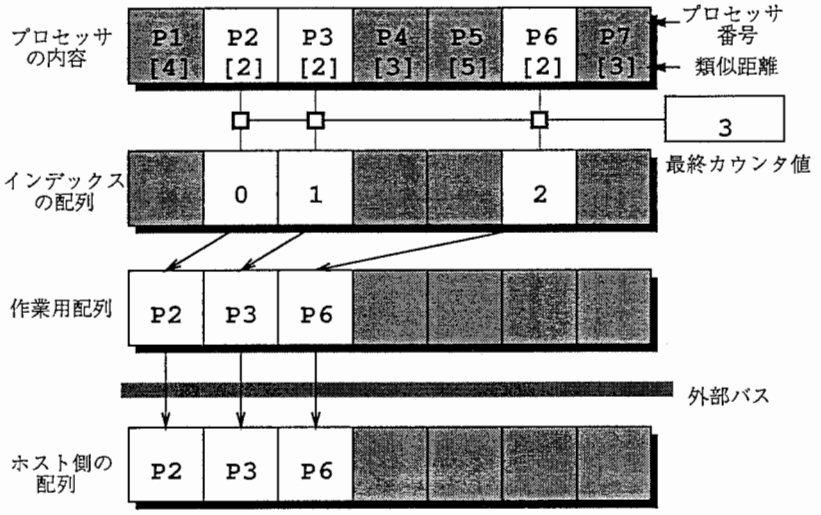

図 4.2: 収集プログラム (改良後)

4.4 収集プログラムの計算量の比較

収集プログラムの計算量を表 4.1、表 4.2に示した。改良前のプログラムは用例数に、改良後のプログラムはマーク付き用例数に依存する。

表 4.1: 計算量 (改良前)

処理 A.1	$O(n)$

n : 用例数

表 4.2: 計算量 (改良後)

処理 B.1	$O(1)$
処理 B.2	$O(1)$
処理 B.3	$O(n')$
処理 B.4	$O(n')$

n' : マーク付き用例数

図 4.3〜 4.6は処理 B.1 〜処理 B.4 における、図 4.7は処理 B 全体における、マーク付き用例数と処理時間との関係 (実測値) を示している。 1000 件の用例を用いた類似検索データで計測した場合、マーク付き用例数は高々 30 個程度であり平均は 5.8 個程度であった。

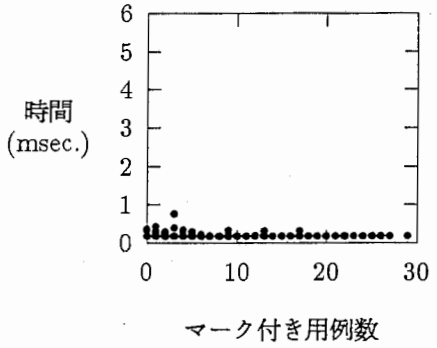

図 4.3: マーク付き用例数と時間の相関 (処理 B.1)

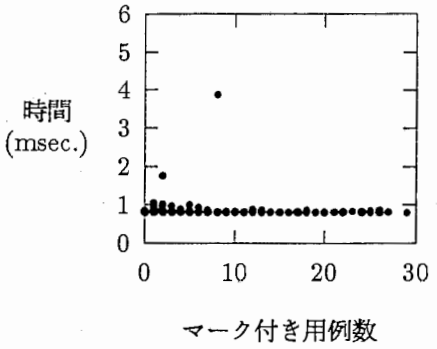

図 4.4: マーク付き用例数と時間の相関 (処理 B.2)

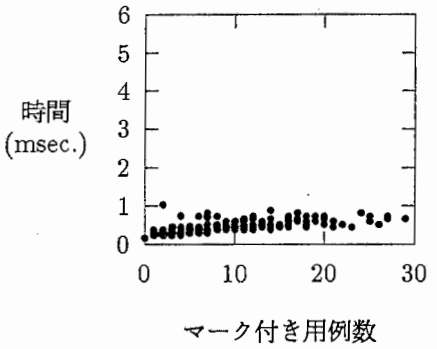

図 4.5: マーク付き用例数と時間の相関 (処理 B.3)

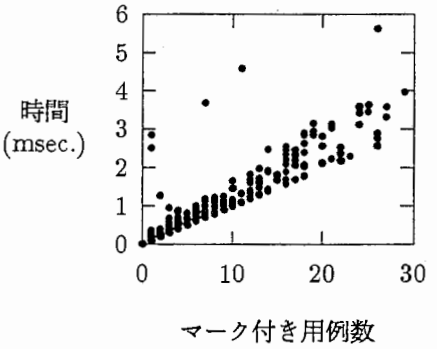

図 4.6: マーク付き用例数と時間の相関 (処理 B.4)

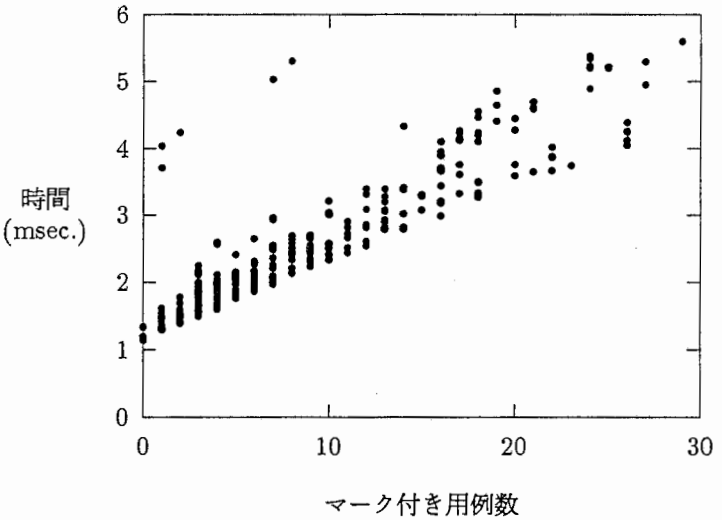

図 4.7: マーク付き用例数と時間の相関 (処理 B 全体)

第 5 章

実験 (プログラム改良の効果)

この節では、第 2 章で述べた 2 つのプログラムを同一のデータに対して適用した結果を示す。

5.1 実験条件

5.1.1 使用計算機の諸元

表 5.1に実験に使用したマシンの仕様を示す。プログラムはコネクションマシンのために設計されたデータ並列機能を持つ C 言語の拡張版である C* で記述した。

表 5.1: ハードウエアの仕様

マシン名	CM-2
プロセッサ数	8192†
メモリ容量	32KB/ プロセッサ
FPU	256 個 (単精度 32 ビット)
フロントエンド	SPARC System 300

†実装されている 16K のうち 8K のみを使用した。

5.1.2 データ仕様

ATR 対話データベース [1] の『国際会議申し込みに関する対話のドメイン』から抽出した「A の B」形式の用例を、以下のように 4 つの検索タイプごとに分類し、各々に対して用例 1000 件と入力 100 件を用意した。

- **見出し完全一致用**
  全入力が用例と見出し完全一致するデータセット。
- **類語コード完全一致用**
  全入力が用例と類語コード完全一致するデータセット。
- **類似検索用**
  全入力が類似検索を必要とするデータセット。
- **混在検索用**
  上記の 3 タイプのデータを混在させたもの。

これらのデータセットで、用例数を 100 から 1000 まで 100 ごとに変化させながら、1 件当りの平均時間を測定した。

5.2 改良前後の結果

図 5.1は改良前後のプログラムによる処理時間を比較したものである。この図は、改良前に用例数に比例して増加していた処理時間が、改良後は用例数に依存せず一定になったことを示している。用例収集を改良することで繰り返し処理を排除し、CM-2 の各要素プロセッサとフロントエンド・コンピュータ間の低速な通信を最低限に抑えることで大幅な性能向上を達成した。

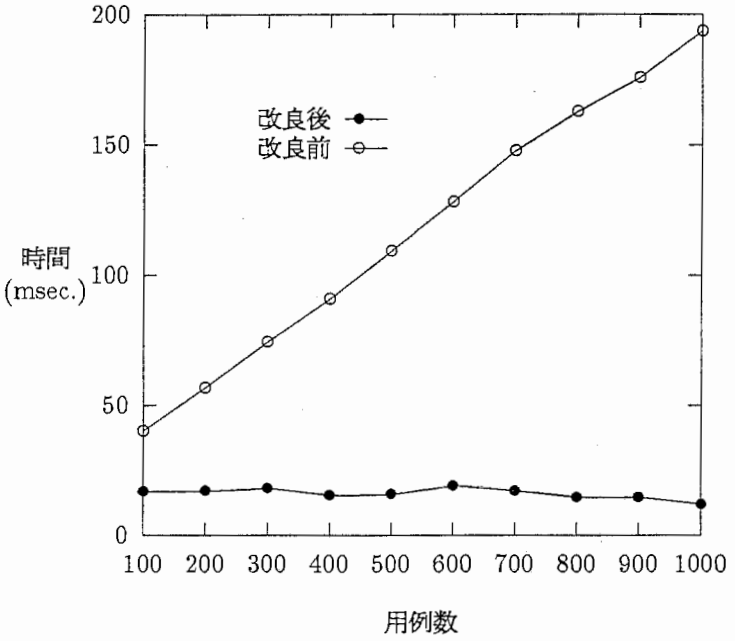

図 5.1: 改良前後の比較

図中の改良後のプログラムの処理時間は約 20 ミリ秒程度であるが、これは、超並列連想プロセッサ IXM2 の性能に匹敵するものである。しかも、ほぼ定数時間、すなわち用例数の多少に関わらず一定とみなせる挙動を示している。

以降の節では、前記のデータセットごとに実際の実験結果のグラフを提示する。グラフ内の項目の内、「マーク付与」は図 4.1の【処理 3.1 ～ 3.3】のマーク付与、及び、意味距離計算に必要な実行時間の総計であり、「用例収集」は【処理 3.4】の用例の収集に必要な時間を表している。

5.2.1 見出し完全一致検索での詳細な比較

見出し完全一致検索ではマーク付与に費やされる時間が非常に小さいため、収集時間が支配的である。

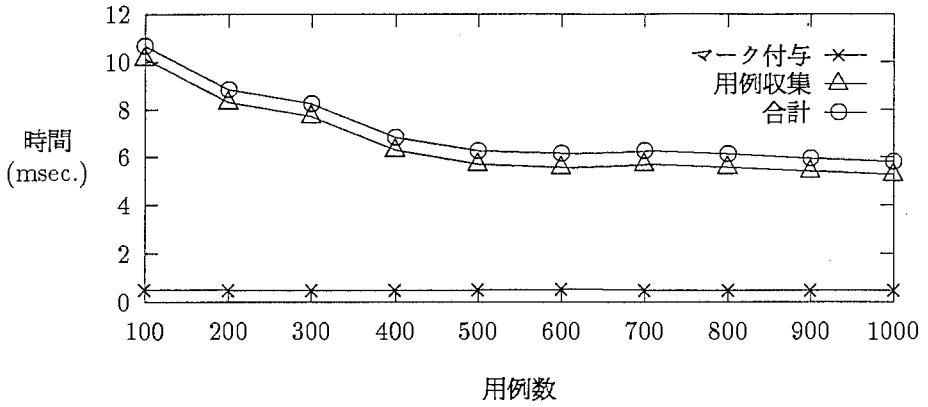

図 5.2: 見出し完全一致検索 (改良後)

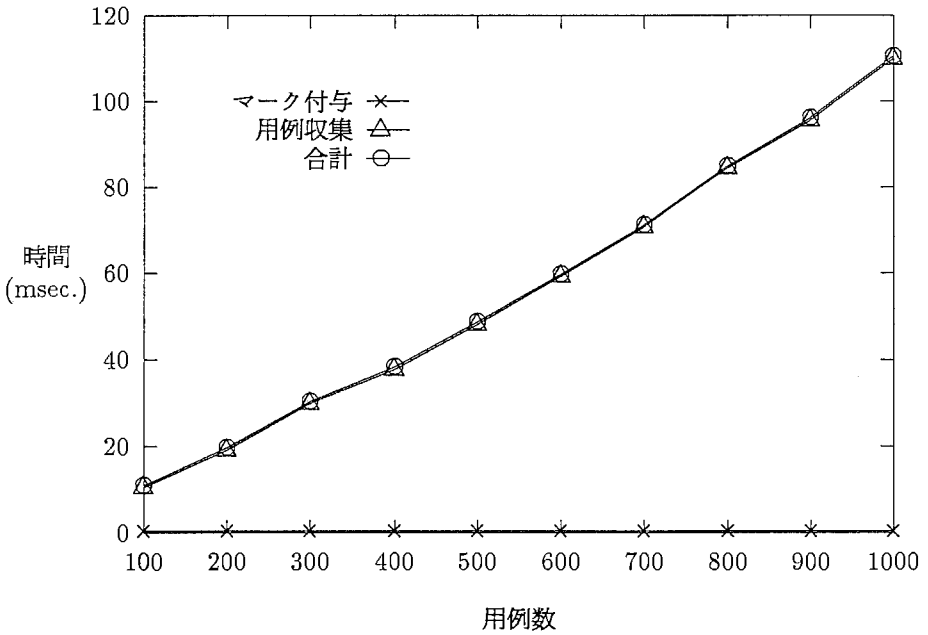

図 5.3: 見出し完全一致検索 (改良前)

5.2.2 類語コード完全一致検索での詳細な比較

見出し完全一致検索とほぼ同様の結果である。マーク付与に比較的時間がかかるのは類語コード完全一致の処理が見出し完全一致検索が失敗した場合に実行されるため、これらの処理時間が累積されているためである。ここでもまだ収集時間が支配的である。

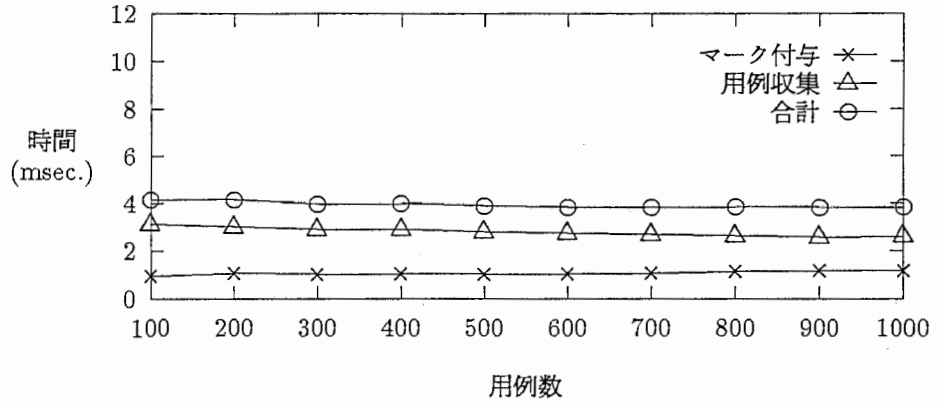

図 5.4: 類語コード完全一致検索 (改良後)

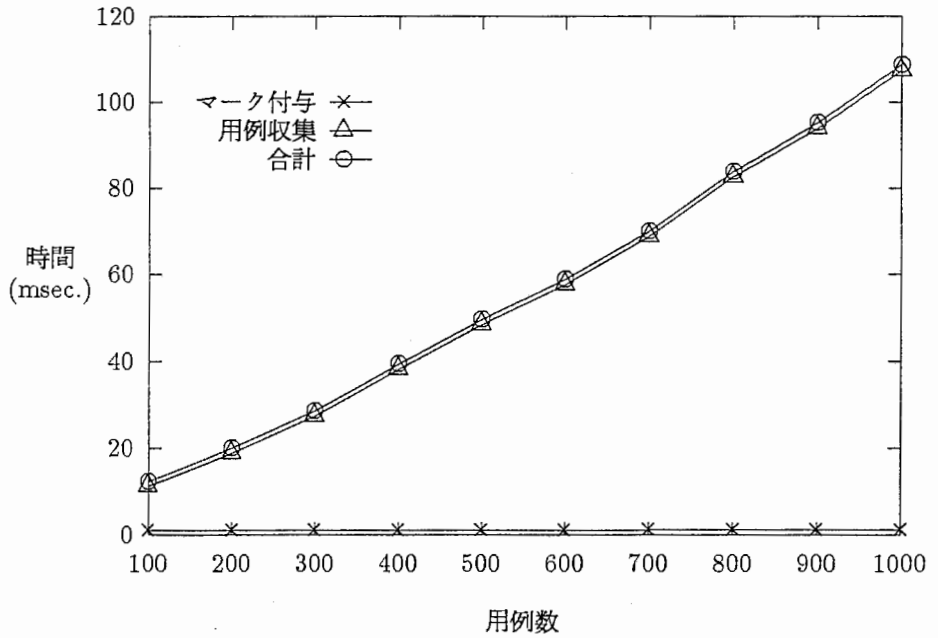

図 5.5: 類語コード完全一致検索 (改良前)

5.2.3 類似検索での詳細な比較

類似検索ではマーク付与時間が支配的になった。これは、類似検索ではマーク付与の処理が比較的複雑になっているためである。

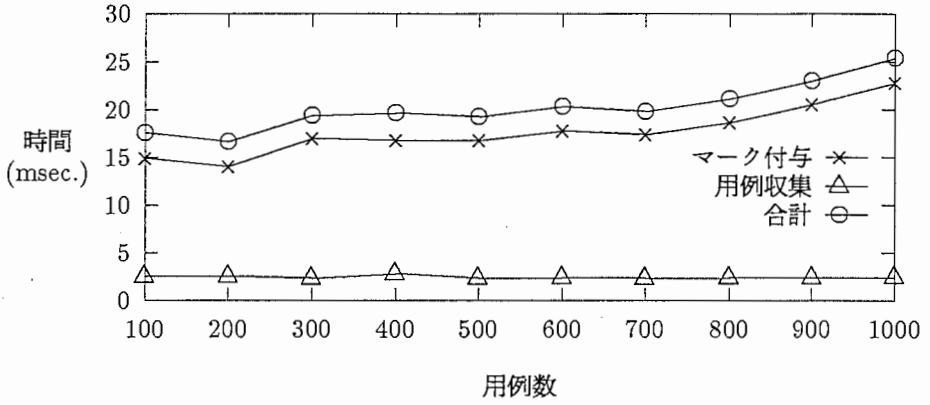

図 5.6: 類似検索 (改良後)

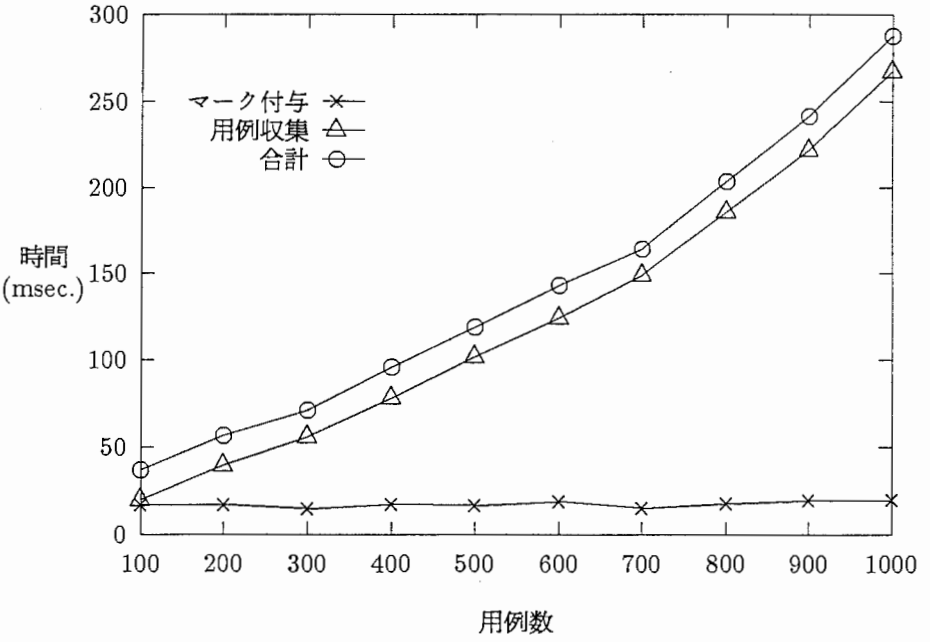

図 5.7: 類似検索 (改良前)

5.2.4 混在検索での詳細な比較

混在型では、平均的な挙動が観察できる。マーク付与時間が変動しているのは、入力ごとに、(1) 高速な見出し完全一致、及び、類語コード完全一致検索と、(2) 比較的遅い用例検索との割合いが変化するためである。

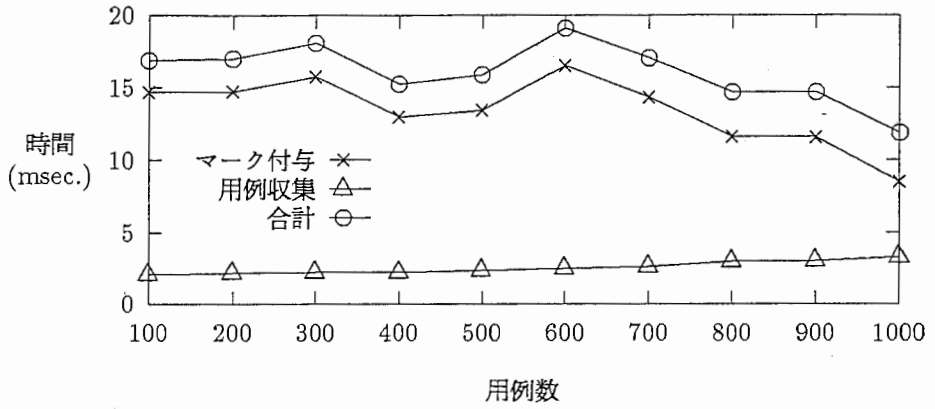

図 5.8: 混在検索 (改良後)

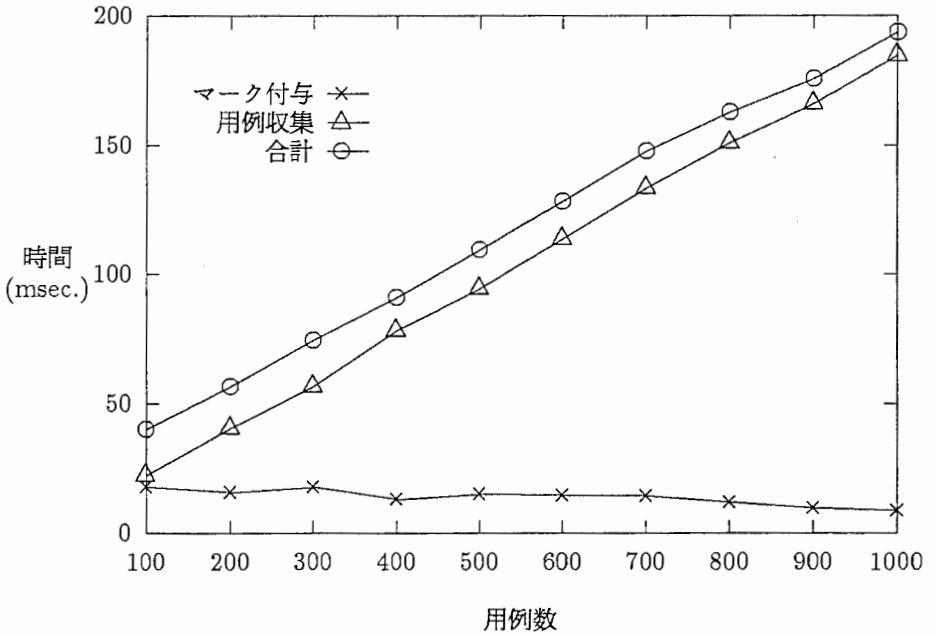

図 5.9: 混在検索 (改良前)

第 6 章

大規模用例に対する挙動

ここでは、CM-2 の物理プロセッサ数を越えるような大規模データを用いた場合に、改良後のプログラムがどのような挙動を示すかを検証する。用例データとして、朝日新聞の記事データから抽出された「A の B」形式の大規模な名詞句データ (10 万件) [8] を使用する。

6.1 大規模用例を用いた実験の概要

これまでは、用例数 1000 件以下のデータを使用して実験を行なってきた。実験に用いた CM-2 は 8192 プロセッサが実装されているので、用例を物理プロセッサに 1 対 1 で割り当てることができたわけだが、10 万件の用例データに対しては用例が物理プロセッサ数を越えるため、何らかの工夫が必要になる。このような問題に対して、CM-2 では仮想プロセッサ (VP) を提供している。仮想プロセッサとは、各物理プロセッサが、複数の仮想的なプロセッサをシュミレートすることである。

- **仮想プロセッサ比 (VP 比)**
  仮想プロセッサ比とは、物理プロセッサが何台分の仮想プロセッサをシュミレートしているかを示す指標である。仮想プロセッサ比は次のように計算することができる。

$$\text{仮想プロセッサ比} = \frac{\text{仮想プロセッサ数}}{\text{物理プロセッサ数}}$$

このように、CM-2 では物理プロセッサ数より大きなデータが必要な場合に、システムが自動的に仮想プロセッサを割り当てるので、プログラム上は物理プロセッサ数を特に意識しなくて済むようになっている[1]。しかし、実際は物理プロセッサが VP 比に応じた処理を繰り返し実行することと等価であるため、処理時間は VP 比に比例して増加する。

[1] CM-2 の場合、仮想プロセッサ数は物理プロセッサ数の整数倍にしなければならないなどの制約も多い。

6.2 大規模データの性質

図 6.1は、入力用例 100 件中の各検索タイプの頻度を表している。1000 件のデータセットについてはこのような性質を説明しなかったが、ほぼ同等の傾向を示している (詳細は [3] 参照)。一般的に、用例が増加すると特定の入力用例と Exact-match する確率は高くなる。

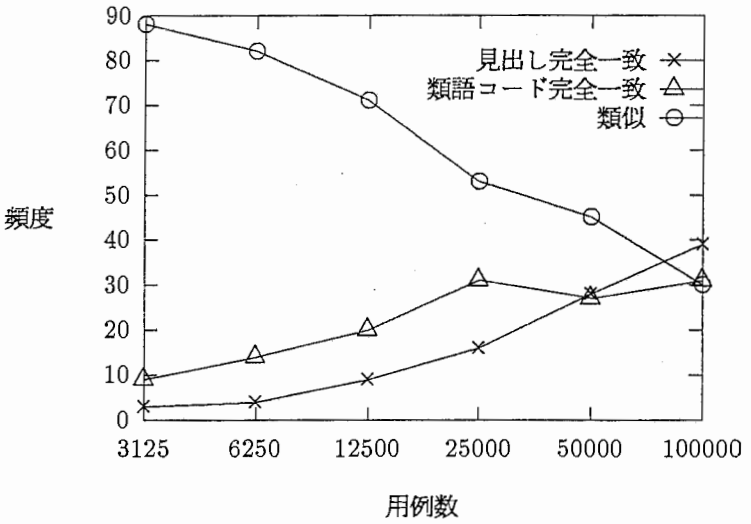

図 6.1: 用例 10 万件での検索タイプの割合

6.3 大規模用例での結果

図 6.2が 10 万件の用例データを使用した場合の処理時間である。用例数と VP 比の関係は表 6.1参照。処理時間が VP 比に必ずしも比例しないのは、前節で示した通り用例数の増加に伴って、見出し完全一致、及び、類語コード完全一致の確率が上がり、処理が比較的複雑な類似検索の割合が減るためである。これをまとめると次のようになる。

- 処理時間の上限は、VP 比に比例して増加する。
- しかし、データの性質の変化により、用例が増えるに従って実質的な処理時間は減少する。

表 6.1: 用例数と VP 比の関係

用例数	3125	6250	12500	25000	500000	100000
VP 比	1	2	4	8	16	32

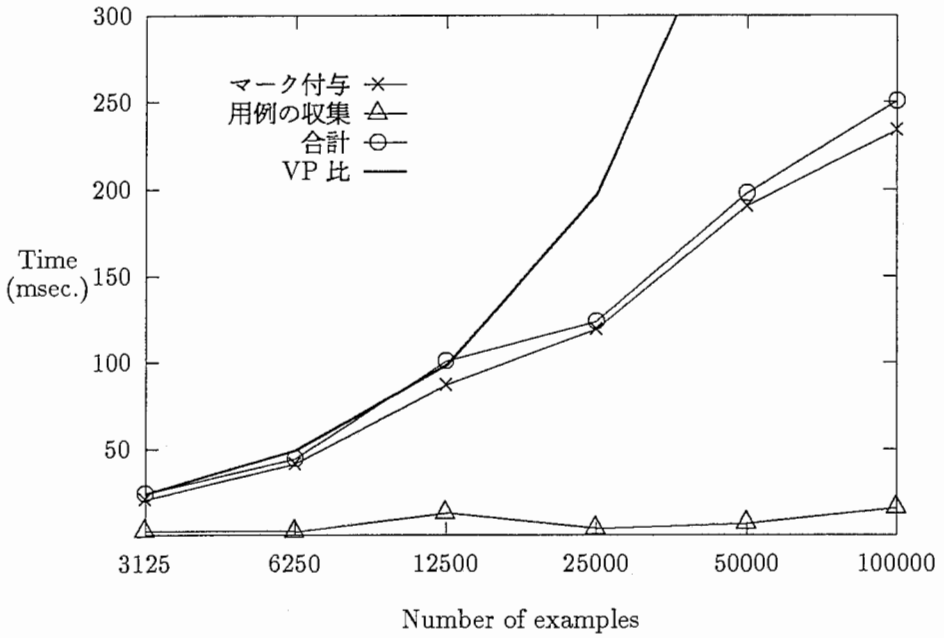

図 6.2: 用例 10 万件での混在検索

6.4 処理時間と VP 比の関係

これまでの実験から、以下のような用例数と VP 比の関係が明らかになった。

- **用例数が物理プロセッサ数以内の場合** (VP 比 = 1)
  処理時間は用例数に関係なくほぼ一定である (約 20 ミリ秒)。
- **用例数が物理プロセッサ数を越える場合** (VP 比 > 1)
  処理時間は、VP 比に比例して増加する (VP 比 * 20msec)。

10 万件の場合、約 $20 * 32 = 640$ ミリ秒以下で処理することが可能である[2]。仮に CM-2 の最大構成の 64K 個の要素プロセッサを利用した場合、100 万件の用例検索を約 $20 * 34 = 680$ ミリ秒以下で処理できる計算になる。

[2]実際は約 250 ミリ秒程度で済んでいる。

第 7 章

より高速な検索に向けて

収集プログラムの改良により、今まで影響の少なかった用例へのマーク付与のための時間がボトルネックになってきた (5.2.3参照)。この章ではこの問題に対する対策をいくつか検討する。

7.1 実数の型宣言の指定

今回実験に使用した CM-2 に実装されている FPU は単精度 32 ビットのものであり、プログラム中で定数を使用している部分は、そのままではコンパイル時に一旦倍精度型 (double) に変換されソフトウエアシュミレーションで実行され非常に遅くなってしまう。この問題を回避するためにプログラム中で明示的に定数に対して単精度 (float) 型の宣言[1]を行なうことで実際に FPU の性能を引き出すことが可能になる。

7.2 データのパッキング

CM-2 はビットシリアル計算機であるため、各プロセッサは 1 度に 1 ビットづつしか処理することができないため、データのビット数に比例して演算時間、及び、転送時間が増加する。現在のプログラムでは全ての属性値が整数型 (int) で宣言されているが、これを最小限必要な有効ビット数まで切り詰めれば処理速度が向上することが予想される[2]。しかし、今回は前報のプログラムとの比較することを重視して改良は行なわなかった。一方、この問題はビットシリアル型計算機のアーキテクチャに依存しているため、現在主流であるワード指向計算機ではあまり問題とならず、プログラムも複雑になり、アンパックにも余分な時間がかかるため、CM-2 以外ではメモリ領域に余裕がない場合などを除くと十分な効果は期待できない。

7.3 意味距離計算のテーブル化

類似検索は、他の見出し完全一致検索、及び、類語コード完全一致検索と比べて比較的複雑処理を行なうため、この類似検索の効率を上げることは有用である。この類似検索の中でも、類語コード同士の意味距離計算が処理の大部分を占めている。そこで、意味距離計算を類語コード[3]のテーブルルックアップ操作とみなし、あらかじめ 1000*1000 の類語コードのルックアップテーブルを作成しておくことにより意味距離計算の高速化が可能である。実際に、逐次版で実験を行ないその有効性を確認[4]しているが、ルックアップテーブルが必要とするメモリ領域がかない大きいため、CM-2 ではメモリ (32KB) の制約により実現は困難である。

---

[1] 例えば "1.0F" のように指定する。

[2] 実際文献 [3] では超並列連想プロセッサ IXM2 上でこの最適化を行なっている。

[3] 類語コードは 0 ～ 999 の整数で表現されている。

[4] SS10 上で 10 万件の用例データを使用した場合、処理時間は約 1/2 になった。

第 8 章

おわりに

我々は、SIMD 方式の計算機 (CM–2) の欠点である低速な通信を最小限度に押えた高速検索アルゴリズムを提案した。このアルゴリズムによって、検索が約 20 ミリ秒 * 仮想プロセッサ比 (VP 比) 以下で実行できることを確認した。

用例主導機械翻訳においては 1 文の翻訳中に何千回も用例検索が呼び出されることがある。リアルタイム処理を実現するにはさらに高速化が必要となるが、上記の結果を元に SIMD 方式の計算機の性能から用例検索の性能を概算することが可能となった。

今後は、SIMD 方式以外の並列計算機に対しても実験を行ない、用例検索に最適な方式を求めたい。

謝辞

CM–2 の利用を許可していただいた、エイ・ティ・アール人間情報通信研究所の関係各位に深謝致します。

参考文献

[1] 江原 暉将, 小倉 健太郎, 篠崎 直子, 森本 逞, 樽松 明. 『電話またはキーボードを介した対話に基づく言語データーベース ADD の構築』. 情報処理学会論文誌, Vol. 33, No. 4, pp. 448–456, 1992.

[2] Furuse, O. and Iida, H. "Cooperation Between Transfer and Analysis in Example-Based Framework". *Proc. of Coling '92*, pp. 645–651, 1992.

[3] 大井 耕三, 隅田 英一郎, 飯田 仁. 『超並列連想プロセッサ IXM2 を使った用例主導機械翻訳の並列化手法』,. Technical Report TR-I-0318, ATR technical report, 1993.

[4] Sumita, E. and Iida, H. "Example-Based NLP Techniques - A Case Study of Machine Translation". *Statistically-Based Natural Language Processing Techniques-Papaers from the 1992 Workshop*, 1992.

[5] Sumita, E. and Iida, H. "Example-Based Transfer of Japanese Adnominal Particles into English". IEICE TRANS. INF. & SYST., Vol. E75-D, No. 4, pp. 585–594, 1992.

[6] Sumita, E., Furuse, O. and Iida, H. "An Example-Based Disambiguation of Prepositional Phrase Attachment". *Proc. of TMI-93*, pp. 80–91, 1993.

[7] Sumita, E., Oi K., Furuse, O., Iida, H., Higuchi, T., Takahashi, N. and Kitano, H. "Example-Based Machine Translation on Massively Parallel Processors". *Proc. of IJCAI '93*, Vol. 2, pp. 1283–1288, 1993.

[8] 田中康仁. 『語と語の関係解析用資料 - 朝日新聞記事データ (84 日分)- - "の" を中心とした -』, 1991.

付録 A

用例検索のプログラム (改良後)

C* で記述された用例検索のプログラムを掲載する。

```
#include   <stdio.h>
#include   <cstimer.h>
#include   <cscomm.h>

#define      XXX       1000 /* 未定義語 */
#define      YYY       1001 /* コンピュータ用語 */
#define      MAXEXM      1971 /* 入力データ最大数 */
#define      MAXINP      8192 /* 用例データ最大数 */

shape       [MAXEXM]pattern; /* ＳＨＡＰＥの定義 */
int:pattern    eid = −1, eam, eac, eas, eno, ebm, ebc, ebs, flg, idx, result;
float:pattern    dis;

int     iid[MAXINP], iam[MAXINP], iac[MAXINP], ias[MAXINP], ino[MAXINP], ibm[MAXINP],
ibc[MAXINP], ibs[MAXINP];
float       wac[MAXINP], was[MAXINP], wno[MAXINP], wbc[MAXINP], wbs[MAXINP];
int       nex, nin, sav_eid[1000];

/**************************************
 用例ファイルの読み込み。
**************************************/
read_example_file( file_name )
char *file_name;
{
    FILE *fp;

    if( (fp = fopen( file_name, "r" )) == NULL ) {
     fprintf( stderr, "ファイル %s がオープンできませんでした。 \n", file_name );
     exit(−2);
    }

    nex = 0;          /* 用例データカウンタのクリア */
    while( (fscanf( fp, "%d %d %d %d %d %d %d %d",
```

```
        &[nex]eid, &[nex]eam, &[nex]eac, &[nex]eas, &[nex]eno, &[nex]ebm, &[nex]ebc, &[nex]ebs
)) ≠ EOF ) {
     nex++;       /* 用例データのカウント */
     if( nex ≥ EXM_MAX ) {
         fprintf( stderr, "用例データが多過ぎます.\n" );
         exit(−3);
     }
    }
    fclose( fp );
}

/*****************************************
 入力ファイルの読み込み。
*****************************************/
read_input_file( file_name )
char *file_name;
{
    FILE *fp;

    if( (fp = fopen( file_name, "r" )) == NULL ) {
     fprintf( stderr, "ファイル %s がオープンできませんでした。 \n", file_name );
     exit(−4);
    }

    nin = 0;            /* 入力データカウンタのクリア */
    while( (fscanf( fp, "%d %d %d %d %d %d %d %d %f %f %f %f %f",
          &iid[nin], &iam[nin], &iac[nin], &ias[nin], &ino[nin], &ibm[nin], &ibc[nin], &ibs[nin],
          &wac[nin], &was[nin], &wno[nin], &wbc[nin], &wbs[nin] )) ≠ EOF ) {
     nin++;       /* 入力データのカウント */
     if( nin ≥ INP_MAX ) {
         fprintf( stderr, "入力データが多過ぎます.\n" );
         exit(−5);
     }
    }
    fclose(fp);
}

/*****************************************
 一致した用例のＩＤを格納する。
*****************************************/
display_1(f)
int f;
{
    int k, done = −1, p = 0;
    int total;

    CM_timer_start(4);

    with (pattern) {
```

```
    where ( flg == f ) {    /* マーク付与された用例を集める */
        total = +=(int:pattern)1;
        idx = enumerate( 0, CMC_upward, CMC_exclusive, CMC_none, CMC_no_field );
        [ idx ]result = eid;
    }
    }
    for (k = 0; k < total ; k++) {
    if ([k]result ≠ done) { /* ホスト配列に転送する */
        done = sav_eid[p++] = [k]result;
    }
    }
    CM_timer_stop(4);
}

/*****************************************
  最小の距離を持つ用例を格納する。
*****************************************/
display_2(d)
float d;
{
    int k, done = −1, p = 0;
    int total;

    CM_timer_start(4);
    with (pattern) {
    where ( dis == d ) {    /* 最小値を持つ用例を集める */
        total = +=(int:pattern)1;
        idx = enumerate( 0, CMC_upward, CMC_exclusive, CMC_none, CMC_no_field );
        [ idx ]result = eid;
    }
    }
    for (k = 0; k < total ; k++) {
    if ([k]result ≠ done) { /* ホスト配列に転送する */
        done = sav_eid[p++] = [k]result;
    }
    }
    CM_timer_stop(4);
}

/*****************************************
  用例検索処理
  *****************************************/
main( argc, argv)
int argc;
char *argv[];
{
    int j, jtmp, match_flg;
    float min;
```

```
with (pattern) {

if (argc ≠ 3) {
    fprintf(stderr, "使用法: %s <用例ファイル> <入力ファイル> \n", argv[0]);
    exit(−1);
}

read_example_file( argv[1] );   /* 用例データファイルの読み込み */
read_input_file( argv[2] ); /* 入力データファイルの読み込み */

CMC_timer_clear(1);   /* 意味距離計算時間計測用 */
CMC_timer_clear(2);   /* マーク付与時間計測用 */
CMC_timer_clear(3);   /* 意味距離最小値取得時間計測用*/
CMC_timer_clear(4);   /* 最小用例収集時間計測 */
CMC_timer_clear(5);   /* 用例検索計測用 */

CMC_timer_start(5);
j = 0;      /* カレント入力データのインデックス */
while (j < nin) {

    CMC_timer_start(1);
    dis = 0.0;   /* 意味距離とフラグの初期化 */
    flg = 0;
    match_flg = 0;

    where (iam[j] == eam && ias[j] == eas && ino[j] == eno &&
      ibm[j] == ebm && ibs[j] == ebs) {
     flg = 1;   /* 見出しが一致すればマークを付与する */
    }
    CMC_timer_stop(1);

    CMC_timer_start(2);
    match_flg = >?= flg; /* 一致した用例があるかを検査する */
    CMC_timer_stop(2);

    if (match_flg == 1) {
     display_1(1); /* 一致していれば用例を収集する */
     do {   /* 次のＩＤまでスキップする */
      j++;
     } while (iid[j] == iid[j − 1]);
     continue;
    }
    jtmp = j;

    CMC_timer_start(1);
    do {
     where (iac[j] == eac && ias[j] == eas && ino[j] == eno &&
       ibc[j] == ebc && ibs[j] == ebs) {
      flg = 1; /* 類語コードが一致すればマークを付与する */
```

```
 }
 j++;
} while (iid[j] == iid[j - 1]);
CMC_timer_stop(1);

CMC_timer_start(2);
match_flg = >?= flg; /* 一致した用例があるかを検査する */
CMC_timer_stop(2);

if (match_flg == 1) {
 display_1(1); /* 一致していれば用例を収集する */
 continue;
}
j = jtmp;
CMC_timer_start(1);
do {
 dis = 0.0;

 where (ias[j] ≠ eas) {
  dis += was[j];    /* Aの接尾語の見出し */
 }
 where (ino[j] ≠ eno) {
  dis += wno[j];    /* 「〜の」の見出し */
 }
 where (ibs[j] ≠ ebs) {
  dis += wbs[j];    /* Bの接尾語の見出し */
 }
 where (iac[j] == XXX || eac == XXX) {
  dis += wac[j];    /* Aの類語コード */
 }
 else where (iac[j] == eac) {
 }
 else where ((int)(iac[j] / 10) == (int:pattern)(eac / 10)) {
  dis += wac[j] / 3.0;
 }
 else where ((int)(iac[j] / 100) == (int:pattern)(eac / 100)) {
  dis += wac[j] * 2.0 / 3.0;
 }
 else {
  dis += wac[j];
 }

 where (ibc[j] == XXX || ebc == XXX) {
  dis += wbc[j];        /* Bの類語コード */
 }
 else where (ibc[j] == ebc) {
 }
 else where ((int)(ibc[j] / 10) == (int:pattern)(ebc / 10)) {
  dis += wbc[j] / 3.0;
```

```
        }
        else where ((int)(ibc[j] / 100) == (int:pattern)(ebc / 100)) {
         dis += wbc[j] * 2.0 / 3.0;
        }
        else {
         dis += wbc[j];
        }
        j++;
       } while (iid[j] == iid[j - 1]);
       CMC_timer_stop(1);

       CMC_timer_start(3);
       min = <?= dis;        /* 最小距離を持つ用例の検索 */
       CMC_timer_stop(3);
       display_2(min);       /* 最小距離を持つ用例の収集 */
    }

    CMC_timer_stop(5);

    CMC_timer_print(1);    /* 実行時間の表示 */
    CMC_timer_print(2);
    CMC_timer_print(3);
    CMC_timer_print(4);
    CMC_timer_print(5);

   }
}
```