細粒度並列処理によるダブル配列言語モデルの構築高速化

石井瑛彦 *1　芳賀 駿平 *1　竹中孝介 *1　大隅賢二 * *2　山本 幹雄 *1
*1筑波大学大学院 システム情報工学研究科 *2筑波大学情報学群

1 はじめに

コンパクトで高速な ngram 言語モデルの実装法としてダブル配列 (Aoe, 1989) を用いた言語モデルである DALM(Double Array Language Model) (Norimatsu et al., 2016) が提案されている。DALM は高速さを保ったままモデルサイズを大幅に削減することに成功しているが他の実装法に比べ構築に膨大な時間がかかる (Nikolay et al., 2016)。高速化の工夫として入力であるトライ木を小さな部分木に分割し、それぞれの部分木に対応する DALM を並列に構築するという手法が取られているが、細かく分割するとモデルのサイズが大きくなってしまう問題がある。また、分割数以上に並列数を上げることができないために、構築アルゴリズムの改良の自由度が低い。

本稿では、分割数に依存しない並列構築の基本アルゴリズムを提案し、DALM 構築アルゴリズム設計の自由度を高めることを目指す。具体的には、入力を分割するという粗粒度の並列化に対し、一つ一つの DALM の構築を細粒度で並列化する手法を中心に、ノード配置順のランダム化と部分転置ダブル配列 (芳賀他, 2016; 竹中他, 2017) の併用による比較的高速な並列化手法を検討する。

2 DALM: ダブル配列言語モデル

2.1 ダブル配列

ダブル配列はトライ (Fredkin et al., 1960) の遷移表の各行を値を持つ要素が重ならないようにずらして圧縮するという発想で疎になりがちな遷移表を Base, Check と呼ばれる 2 本の密な配列で表現した、コンパクトで高速なデータ構造である (Aoe, 1989)。図 1 はその構築方法を表している。

まず遷移表の各行を要素が縦に重ならないように横にずらして next と呼ばれる配列に押し込む。さらに next 配列を正しく遷移するために各要素の遷移元ノード (親ノード) の情報を check 配列で保持し、各行のずらし幅を offset 配列として保持する。さらにこの offset 配列を削るためその値で next 配列中で対応するノード ID を書き換えこれを新たに base 配列とする。次に check 配列の値をノード ID に対応する next 配列の index で書き換え新たな check 配列とする。こうして出来た base と check の 2 本の配列で遷移表を表現したものがダブル配列である。

2.2 ダブル配列言語モデル

ダブル配列言語モデル (以降 DALM) は ngram 言語モデルの実装にダブル配列を適用した言語モデルである。トライで表現された ngram 言語モデルは受け取った単語列をキーとしてその生起確率が格納されている場所にアクセスしそれを返す。しかし学習したデータの中にその単語列が

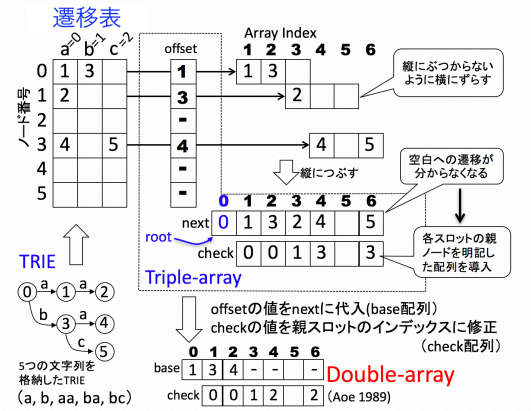

図 1: TRIE の疎行列表現とダブル配列

ない場合もあるのでその際に入力された n-gram の確率を (n-1)-gram の確率に係数をかけて近似した値で代用する。その際に用いる係数をバックオフ係数と呼び、各 n-gram は生起確率と一緒にバックオフ係数を保持している。

これら 2 つのパラメータを保持する方法には 2 種類の実装法がある。1 つ目は 2 つのパラメータを base,check 配列中の未使用スロットに格納するという手法 (Norimatsu et al., 2016) でモデル全体のサイズをコンパクトにできるがパラメータを量子化することができない。2 つ目として、量子化を可能とするためにパラメータ用の配列を別に用意する手法 (芳賀他, 2016; 竹中他, 2017) が提案された。この手法ではダブル配列の隙間を利用しないので、できるだけダブル配列に隙間を作らずかつ短くする必要がある。そのために開発された手法が leaf-last 法と部分転置法である。図 1 を見るとノード 2,4,5 のような葉ノードつまり子ノードを持たないノードは base 値を持たないのでわざわざ配列の要素として保持しておく必要がない。leaf-last 法ではそれらを配列の最後にまとめて配置し、ベース値の必要ない部分を打ち切ることで base 配列の長さを短縮する。しかし base 配列の長さが子ノードの数が非常に多い数%の内部ノードが配置された時点でほぼ決まってしまうためそのままでは効果が薄かった (以降 “ある内部ノードの子ノード列をダブル配列に配置する” ことを “ノードを配置する” と表現する)。これに対して図 2 のように特に子ノードの多い上位数千個の内部ノードを転置することで配列長の増大を抑え leaf-last 法の効果を高める方法が部分転置法である。

3 部分転置とランダム順配置による高速化

DALM 構築において最も時間がかかる処理は、ノードの子ノード列が他の子ノード列と重ならないようなずらし幅を決める部分である。一般的にはノードをトライの深さ優

*2 現在, 遠鉄システムサービス株式会社

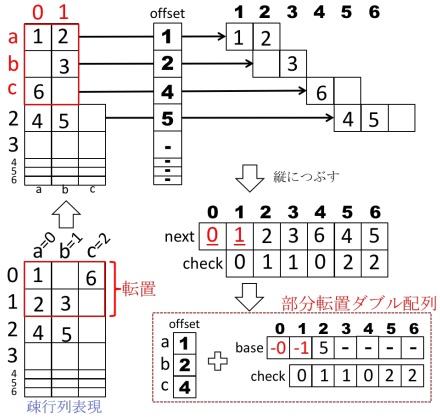

図 2: 部分転置ダブル配列

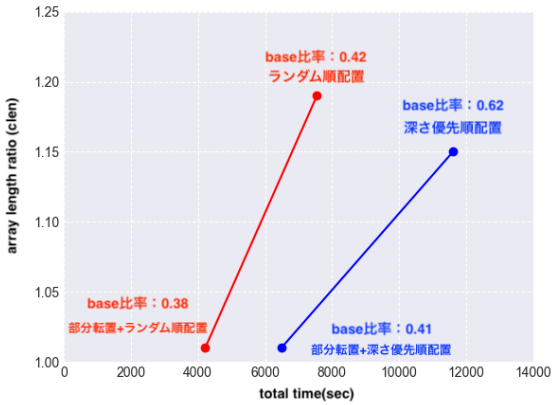

図 3: 深さ優先順とランダム順による構築時間・配列長比率

先順に配置するがトライのサイズが大きくなると配列の隙間が増えてしまいメモリ効率が悪くなるという問題があった (芳賀他, 2016)。ノードを子ノード数降順に配置することで配列充填率をほぼ 100%とすることができるがその場合構築が遅い。配置順序をランダム順にすることで高速化することができるが充填率を下げてしまう。そこでランダム配置順と部分転置を組み合わせることでコンパクトさを保ったまま構築の高速化ができると考えた。

次のような実験を行った。ノード配置の順序は深さ優先順, ランダム順, 子ノード数降順の 3 種類、これに部分転置を行った場合 (転置数は 1000 ノード) とそうでない場合の合計 6 種類の条件で、DALM の構築時間と構築された DALM の配列長を測定した。本稿の 4 節までの実験はすべて Xeon E5-2620v4 2.10GHz,16 コア, メモリ 256GB 上で行った。実験に用いた ngram 言語モデルは、NTCIR-10 (Goto et al., 2013) の 1993 年の特許文データを基に作成した語彙数 619,391,ngram 数 100,000,250(約 1 億) である。配置順序の違いによる効果が出やすいよう 3,4,5-gram について出現頻度が 1 のものはカットオフしてある。

実験結果を図 3 に示す。グラフは横軸に構築時間 (秒)、縦軸に ngram 数に対する check 配列長の比率をとった (隙間なく詰まれば比率は 1.0 となり、隙間があればそれより大きくなる)。残念ながら子ノード数降順は時間が 30 時間以上かかるため、問題外としてこの図には載せなかった。また leaflast 法により base 配列がどれほど削れたかを評価するため各点横に ngram 数に対する base 配列長の比率を記した (base 配列は 1.0 より小さくなり得る)。

実験結果より深さ優先順配置と部分転置を組み合わせることにより 2 つの配列を短くすることができた上に構築時間も短くなった。さらにランダム順配置と部分転置の組み合わせでさらなる高速化に成功しサイズも小さくなった。以降、並列化の検討では部分転置+ランダム順配置を基本とする。

4 細粒度並列処理

4.1 DALM の並列構築: 粗粒度並列処理

従来の DALM では大きなモデルを構築する場合トライを 1 段目のノードで分割してできた小さなトライに対してそれぞれ並列に別々のダブル配列を構築することで構築時間を短縮している (Norimatsu et al., 2016)。

ダブル配列構築にかかる時間は、語彙数を定数とすればトライのノード数 E に対して $O(E^2)$ であるのでそれらを D 個に分割して構築できれば理論上構築時間は $1/D$ になる。さらに、D 個のプロセッサで並列処理すれば $1/D^2$ まで高速化できるはずである。D は一般的に数十なので、これを粗粒度並列化手法と呼ぶ。計算量は最悪時のものなのでもちろんそこまで速くはならないが、特に並列処理の場合、それぞれが完全独立な処理にもかかわらず粒度の不揃いの問題で理想的な速度からさらに遅くなる。これは、分割手法がトライのノードの偏りを考慮しておらず、分割された部分木のノード数に差が生まれてしまうことにより構築完了時間がバラついてしまうから生じる。ある部分木に対する DALM の構築はすぐに終わっても、別の部分木に対する構築に時間がかかれば全体の構築完了時間は最後の構築完了時間となる。構築の遅い部分木を処理している間、すでに構築が終わってしまった CPU は何もしないので、粗粒度並列化従来手法では分割数が増えると構築処理中の CPU 使用率が低くなってしまうという問題があった。そこで本研究では従来のような粗粒度な並列化ではなく構築処理そのものをマルチスレッドで行う細粒度な並列化手法による高速化を提案する。

4.2 細粒度並列アルゴリズム

本手法では構築処理そのものを並列化することで高い CPU 使用率を保ち構築時間の短縮を目指す。従来手法では 2 章で紹介した DALM 構築処理のうち適切なずらし幅を見つけてノードを配置するという処理にかかる時間が全体の大部分を占めていたので本稿ではその部分を並列化する。具体的には、各スレッドは、配置するべき内部ノードを 1 つ割り当てられ、それらが並列に配置処理を行い、配置が終わったらまだ配置されていない別の内部ノードを割り当てられる。これによって CPU の使用率を高い状態で維持できる。数千万から数億ある内部ノード単位の並列化であるため、これを細粒度並列化法と呼ぶ。

ただし、全スレッドは同じ配列に子ノード列を配置していくのでスレッド間には依存関係がある。他のスレッドがどこに配置しようとしているか知りようがないので複数のスレッドが同じ場所に配置してしまう可能性がありそのまま並列化することはできない。とはいえ実際のダブル配列

の構築では高々数十万から数百万の子ノード列を長さ数億の配列上で配置場所を探すので競合は頻発しにくいと考える。それを踏まえたアルゴリズムを Algorithm 1 に示す。

Algorithm 1 並列ノード配置決定アルゴリズム

```
for all parallel トライの内部ノード do
  Confirmed ⇐ FALSE
  while Confirmed ≠ TRUE do
    空き要素リンクをたどり配置場所を探す
    if 子ノード列が配置可能 then
      Critical Section {
        if 子ノード列が配置可能 (再確認) then
          子ノード列配置場所を確保 (マークする)
          空き要素リンクを更新
          Confirmed ⇐ TRUE
        end if
      }
    end if
  end while
end for
※ Critical Section には同時に 1 スレッドのみ侵入可
```

for all の各内部ノード (子を持つノード) についてスレッドが割り当てられる。各スレッドは担当する内部ノードに対して for all 内のアルゴリズムを適用する。おおまかに言うと、処理を (1) 配置場所の探索と (2) 配置の 2 つに分け、(1) のみを並列化し、(2) は 1 度に 1 スレッドしか行えないようにしている。もし (1) で配置可能な場所を発見しても、その前に (2) に入っていた他のスレッドが同じ場所に配置してしまっている可能性があるので (2) で配置可能かどうかを再度確認する。Critical Section 内の配置処理 (配置場所の確保と空き要素リンク更新) を終えたら未配置の内部ノードを担当する。

配置場所を探す際は、配置場所を配列の頭から見て行くのは非効率なので一般に配列の空き要素をリンクでつないでおいて効率的に空き要素をたどる (中村他, 2006)。しかし、細粒度並列化の場合は、リンクをたどっている途中に他ノードの配置によって、リンク先が変化する場合がありえる。特に、メモリ節約のためにリンクを保存する配列を同時に他用途に使う場合はリンクの値が無効となるため注意が必要である。我々の実装では、リンクには次の空き要素の index を正の値で格納し、空き要素が子ノードの配置に使われた場合は負の値でマークする方式を用いた。この場合、確保される前の index の値を負の値で保存すればリンクたどっているときに突然負の値になった場合は正の値になるまでリンクをたどると比較的高速に次の空き要素候補を発見できる。この様子を図 4 に示す。

4.3 粗粒度並列化と細粒度並列化の比較

粗粒度と細粒度の並列化法のそれぞれに対して、構築時間と出来上がったモデルサイズ（ダブル配列の長さ比率）をいくつかのデータに対して比較した。

図 5 は粗粒度と細粒度の並列化法のそれぞれに対して横軸を分割数、縦軸を構築時間とし、並列数毎にプロットしたグラフである。粗粒度並列化法は分割数以上の並列数で構築することができないので、細粒度よりもプロット点が少ない。データは Web 上のニュース記事を用いて作成した

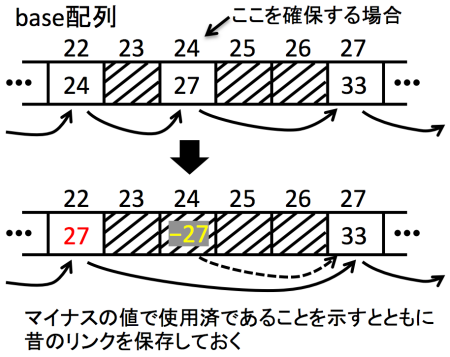

図 4: 空き要素リンク

ngram 数が約 1 億の言語モデルを用いた。なおこれら言語モデルのカットオフは行っていない。実験に使用したマシンは 16 コアだがハイパースレッディングにより 32 スレッドまで並列化できるので分割数および並列数を 2,4,8,16,32 と変化させ実験を行った。転置数は全て 1000 とし転置していない部分の配置順はランダム順としている。共に分割数が増えると構築時間が短縮するが、粗粒度並列化法は 16 分割から 32 分割に増えた際にあまり変わらないかあるいは遅くなってしまっている。これは分割数が大きくなったことで構築時間がばらつき CPU 使用率が低くなってしまったからである。一方細粒度並列化法は最高速度については粗粒度並列化法より遅いものの安定して速くなっている。

図 6 には、さらにデータ数を増やした場合 (ngram 数で 2 億と 5 億) の分割数 (=並列数) に対する構築速度を示す。データを大きくして分割数を増やすと細粒度並列化法は粗粒度並列化法と同程度あるいはより高速になっている。これは粗粒度並列化法では分割数が大きくなるほど構築にかかる時間がバラつき CPU 使用率が低下するためである。大きなデータになれば CPU 使用率をほぼ 100%に維持できる細粒度並列化法が有利になる。

図 7 は分割数 (=並列数) に対する check 配列長,base 配列長の比率 (隙間なく配列が埋まった場合 1.0。隙間ができると 1.0 より大きくなる) を表している。粗粒度法と細粒度法を比較した時にこれらの値はほぼ一致したため (ほぼ完全に重なる)、細粒度法の値でプロットしている。データが大きくなるほど、そして分割数が大きくなるほど check 配列比率が長くなっているのは語彙数の増加により元々配置するのが難しくなっている上に分割されてしまうことで隙間を埋められる子ノード数の少ないノードが分散してしまい全体的に隙間が増えてしまうからである。さらに、leaflast 法が効きにくくなり base 配列も長くなってしまっている。

以上 3 つの図より細粒度の並列化にもかかわらず、提案アルゴリズムは粗粒度並列化とほぼ同等か同等以上の性能を持つことが分かる。

5 細粒度並列化法の応用

粗粒度並列化はトライの分割に基づいているため、分割数より大きな並列化を実行できなかった。今回提案した細粒度並列化手法は、分割数に依存していないため、より柔

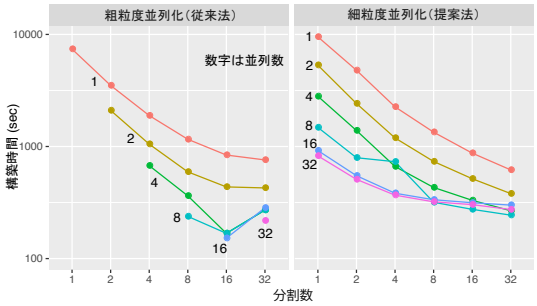

図 5: 分割数・並列数による構築時間の比較 (1 億 ngram)

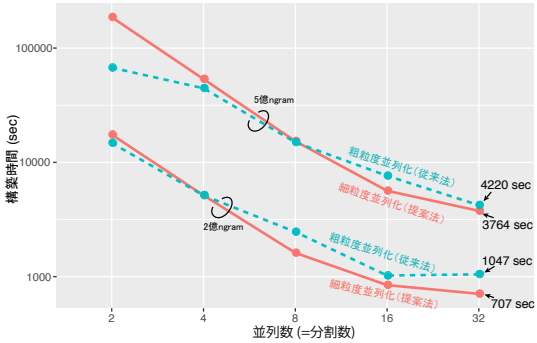

図 6: 分割数=並列数の構築時間の比較:(2 億,5 億 ngram)

軟な構築アルゴリズムを設計できる。本節では、細粒度並列化手法の一つの応用例として、トライの分割は行わずに（最終的に 1 つのダブル配列になる）分割による高速化のメリットを導入する方法を以下に紹介する。

1 つのダブル配列に子ノードを配置していくが、ランダム化された挿入ノードの集合を等分割し、分割毎にダブル配列の区間を区切る。すなわち、新しい分割を挿入する際に、それまでの分割が利用したダブル配列の部分は使わないことにすれば、分割されたダブル配列を作る場合とほぼ同じ高速化のメリットが得られる。しかし、あくまでの 1 つのダブル配列であるため、条件によっては境界を無視して配置することが可能である。子ノード数が少ないノード（比較的簡単に配置できる）に限って大域的にできるだけ左 (index が小さい) への配置を許せば、全体的に配列長を短くすることができる。

分割数を 10,20,50 と変化させ、大域的な配置を許可する子ノード数 (図中の'cn') を 0, 1, 4 とした場合の構築時間と、check 配列・base 配列の ngram 数に対する比のグラフを図 8 に示す（並列数は 32, 転置数は 10,000)。'cn' の変化は図中の「く」の字の折れ線の左から右（構築時間が遅くなる方向）に 0,1,4 である。全体的に、分割数を大きくすると実行速度は速くなるが、配列長は長くなる。しかし、大域的な配置を許可する子ノード数を大きくすれば、実行速度と引き換えに配列長を短くできる。特に子ノード数 1('cn'=1) のノードだけ大域的な配置を許せば、高速性をある程度保ったまま、配列長を短くできることが分かる[1]

[1] 本節の実験で用いた実装方法、転置数、プログラミング言語が前節までの実験と大きく異なるためため前節までの実験結果との比較はできない。

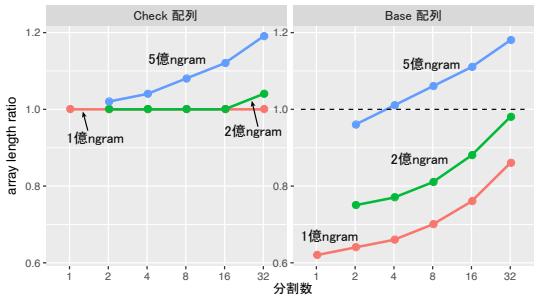

図 7: 配列長比率の比較 (2 億,5 億 ngram)

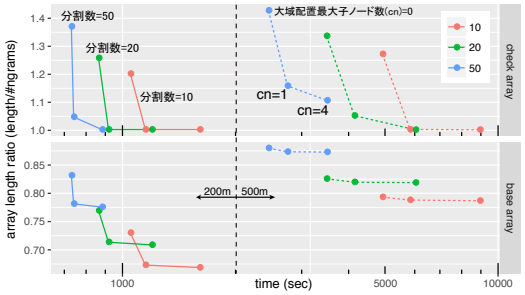

図 8: 子ノード数の少ない内部ノードの大域配置による効果

6 おわりに

本稿では、分割数に依存しない細粒度並列化アルゴリズムを中心に、部分転置とランダム順配置を併用した比較的高速な並列化アルゴリズムを検討・提案した。この基本アルゴリズムは分割に依存しないため、大域的な最適化を同時に行える可能性があり、本アルゴリズムをベースに DALM 構築の高速化を検討していく予定である。

参考文献

Aoe, Jun-ichi (1989).“An efficient digital search algorithm by using a double-array structure,” *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066–1077.

Goto, Isao, Ka Po Chow, Bin Lu, Eiichiro Sumita, and Benjamin K. Tsou (2013). “Overview of the Patent Machine Translation Task at the NTCIR-10 Workshop,” in *10th NTCIR Conference*, pp.260–286.

Norimatsu, J., M. Yasuhara, T. Tanaka, and M. Yamamoto (2016).“A fast and compact language model implementation using double-array structures,” *ACM TALLIP* Vol. 15, No. 4, 27 pages.

Fredkin, Edward (1960). “Trie Memory”,Communications of the ACM, Vol. 3, No. 9, pp. 490–499.

Nikolay Bogoychev, Adam Lopez (2016). “N-gram language models for massively parallel devices.”,*In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 1944–1953, August 7-12.

竹中孝介・芳賀駿平・山本幹雄 (2017).「部分転置ダブル配列を用いた ngram 言語モデルの実装」, 言語処理学会第 23 回年次大会, P13-3, pp. 663–666.

中村康正・望月久稔 (2006).「圧縮ディジタル探索木における辞書情報更新の高速化手法」, 情報処理学会論文誌：データベース, Vol. 47, No. 13, pp. 16–27.

芳賀駿平・谷口正訓・山本幹雄 (2016).「部分転置ダブルアレイを用いた ngram 言語モデルの検討」, 第 30 回人工知能学会全国大会, 4B1-5.