ハイパフォーマンス
コンピューティング 78－7
（1999．10．7）

配列データを共有する並列タスクの実行効率向上のためのアクセス・パターン解析

三田 勝史，朝倉 宏一，渡邉 豊英
名古屋大学大学院 工学研究科 情報工学専攻
{sanda, asakura, watanabe}@watanabe.nuie.nagoya-u.ac.jp

概要

我々は、配列データの依存関係を持つタスク間の並列実行を実現するための手法として、配列データの漸進処理手法を提案している。漸進処理手法は、タスク間の配列操作を漸進的に部分実行する手法で、並列性はタスク間での配列要素の操作順序の類似性に強く依存している。本稿では、漸進処理を適用するための配列アクセス・パターン解析手法を提案する。二重ループ中の二次元配列操作を対象として、配列要素の操作順序の規則性を表現するアクセス・パターンを定義し、タスク間のアクセス・パターン類似度の評価手法について述べる。

Access Pattern Analysis Method for Intertask-dependent Arrays

KATSUSHI SANDA, KOICHI ASAKURA AND TOYOHIDE WATANABE
Department of Information Engineering,
Graduate School of Engineering, Nagoya University
{sanda, asakura, watanabe}@watanabe.nuie.nagoya-u.ac.jp

Abstract

We have proposed the incremental processing method in order to accomplish the parallel executability between tasks with an inter-related array. In this method, each task can access the commonly sharing array incrementally and in parallel through several synchronization points. In the incremental processing, the parallel executability strongly depends on the similarity of access orders for array elements in each task. In this paper, we propose the analysis method of access order for the incremental processing. We define the access pattern to represent the regularity of array access, and discuss the evaluation method for similarity between access patterns.

1 はじめに

プログラム並列化手法において、ループ並列化手法[1, 2, 3]が最も一般的に用いられている。これは、ループ文中の処理を複数のプロセッサに割り当てて並列に実行することで、実行効率を向上させることを目指している。ループ並列化手法は、プログラムを比較的細粒度のタスクに並列化するため、タスク数が膨大になり、またタスク間通信頻度も高くなる。

我々は、逐次実行を前提として作成されたプログラムから、計算機クラスタ環境を対象として並列実行されるタスク群を自動生成する並列化コンパイラを開発中である[4]。並列計算機環境と比較して、計算機クラスタ環境ではタスク起動やタスク間通信時におけるオーバヘッドが大きい。したがって、ループ並列化手法に基づいたタスク生成手法を採用した場合、上記のオーバヘッドにより効率のよい並列実

行を達成できない。そこで、我々は逐次プログラム中の関数手続きを基本単位とした粗粒度タスク生成手法を採用している。関数単位でタスクを生成することで変数空間が局所化されるため、タスク間通信頻度を低く押えることができ、また生成されるタスク数が逐次プログラム中の関数の数以下となるため、プロセス起動オーバヘッドも低く押えることができるという利点がある。

上記手法により生成されたタスク群の並列実行における問題点として、タスク間依存の配列データ操作時の同期処理により、タスクの実行中断状態が発生し、並列実行効率を著しく低下することが挙げられる。我々は、このような状況でタスク間の配列データ操作から並列性を抽出し実行効率を改善するための手法として漸進処理を提案している。本稿は漸進処理を適用する際に必要となるプログラム解析に焦点を絞り議論を進める。以下、2章で漸進処理について説明し、プログラム解析の必要性を述べる。3章では漸進処理のためのプログラム解析として、配列データの操作順序の規則性をアクセス・パターンと定義し、4章ではアクセス・パターンの違いが漸進処理適用時の実行効率に与える影響を考察し、アクセス・パターン類似度の評価方法について議論する。

2 漸進処理手法

本章では、配列データを共有するタスク間での効率のよい並列実行手法として我々が提案している漸進処理について述べる。まず、2.1節で配列データの漸進処理手法の概要を述べる。2.2節で漸進処理適用時のプログラム解析の必要性について議論する。

2.1 配列データの漸進処理

数値計算プログラムには、一つの配列を複数の関数により操作する処理が数多く存在する。また一般に、扱う配列の要素数が非常に大きく、プログラム全体の実行時間に占める配列操作時間の割合が高い。このようなプログラムから並列タスク群を生成しても、同一配列データに対する排他制御により効率のよい並列実行が達成されない状況が発生する。図1は、同一配列データについて真の依存関係 (true dependence) が存在するタスク間の並列実行の様子を示している。producer は配列データを更新するタスク (以後、更新タスクと呼ぶ) で、consumer は配列データを参照するタスク (以後、参照タスクと呼ぶ) である。図中の長方形はタスクが実行されている状態を表現しており、網掛け部分は配列データの操作に対応する。また、左から右に向かって時間経過を表している。参照タスクの配列データ操作は、更新タスクの実行が終了するまで実行中断状態

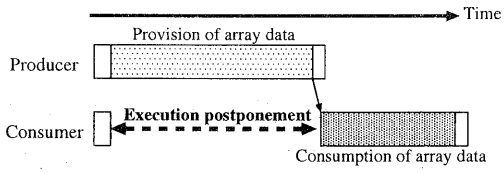

図 1: 配列データを共有するタスク間の並列実行の様子

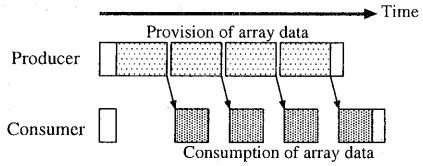

図 2: 漸進処理適用時の実行の様子

となるため、タスクの並列実行はほとんど期待できないことが分かる。

そこで、我々は配列データを共有するタスクの並列性を実現するための手法として、配列データの漸進処理手法を提案している [5]。配列データの漸進処理は、各タスクで配列データの操作順序が類似している場合、配列データ全体を排他的に制御するのではなく、配列データを部分的なデータの集合として捉え、個々の部分データに対して排他制御することでタスク間の配列操作の並列実行を実現する。つまり、配列に対する操作を一箇所の同期点で排他制御していたものを複数の同期点に分割することを意味する。漸進処理を適用した時の実行の様子を図2に示す。更新タスクの実行途中で操作済みのデータを使って、参照タスクを漸進的に部分実行する。これにより一箇所の同期点当りの共有データ量が減少するため、図1の状況と比較して実行中断状態が削減される。

2.2 配列データの操作順序解析

漸進処理の適用は配列データに対する同期点を複数に分割させるため、タスク間通信頻度を増加させる。したがって、タスクでの計算と通信のオーバラップが期待できる場合はタスク間通信のオーバヘッドが隠蔽され、効率のよい並列実行が達成されるが、そうでない場合は漸進処理適用により逆に実行効率が低下する状況が発生し得る。そこで、漸進処理を適用した際の実行効率を事前に評価し、適用すべきか否かを判断する必要がある。漸進処理適用時の実行効率は、タスク間の配列データの操作順序の類似性に大きく依存している。したがって、漸進

処理適用時の実行効率を推定するためには、各タスクでの配列データの操作順序を解析し、タスク間でその類似性を評価することが必要となる。プログラムの並列化において、配列データのアクセス・パターン解析に関する研究は以前から盛んに行われている [6, 7]。しかし、この手法では配列添字式の規則性からイタレーション間やループ間、手続き間でデータ依存関係があるか否かのみが解析可能であり、配列要素の操作順序の類似性の評価には不適切である。そこで、我々は漸進処理のためのプログラム解析として、配列アクセス・パターン解析手法を示す。

3 アクセス・パターン表現手法

本章では、配列データの操作順序をアクセス・パターンとして定義する。まず、3.1節で解析対象を明確にする。3.2節で配列添字式の正規化について述べ、3.3節でアクセス・パターンの表現法について議論する。

3.1 配列データの操作順序の規則性

配列データはほとんどの場合ループ文中で操作され、配列添字式はループ文の実行とともに規則的に変化する。通常、配列添字式はループ変数のみで表現可能であり、既に提案されているループ並列化のためのプログラム解析の中で配列添字式の正規化手法が紹介されている [1, 2, 3]。したがって、配列データの操作順序の規則性を抽出する上で、ループ変数の変化によって引き起こされる添字式の値の変化が重要な情報となる。

本稿では、図3で示すような二重ループ中の二次元配列に対する操作を対象としてプログラム解析を行う。配列添字式はループ変数の線形結合で表現されており、cとdはともに定数であるとする。例えば、`A[10 + i][i + 2*j]` は上記の条件を満たす配列添字式であり、プログラム解析の対象となるが、`A[i][i*j]` のように二次元の添字式がループ変数の線形結合でない場合は対象としない。

```
for(i = lb_i; i <= ub_i; i = i + st_i)
  for(j = lb_j; j <= ub_j; j = j + st_j)
    A[c_0 + c_1 * i + c_2 * j][d_0 + d_1 * i + d_2 * j] = ...;
```

図 3: 対象とするプログラム例

3.2 配列添字式の正規化

図3に示すようなプログラムでの配列データの操作箇所は、規則的な順序にしたがって遷移する。我々は、配列データの操作箇所の規則性を表現したものをアクセス・パターンとして定義する。基本的には、配列添字式は配列データの操作順序の規則性を表現している。しかし、添字式が同一であってもループ変数の変化の違いにより、配列データの操作順序は一意に決定しない。したがって、我々は配列添字式とループ変数の振舞いを考慮して、配列添字式の初期値と、イタレーション毎に発生する配列添字式の値の変化で配列データの操作順序を正規化する。

配列添字式の初期値とは、ループ中で最初に操作される配列要素の添字式の値である。つまり、各ループ変数にその下限値を代入したときの配列添字式の値である。したがって、配列添字式の初期値は一次元、二次元でそれぞれ

$$\begin{aligned} off^1 &= c_0 + c_1 \cdot lb_i + c_2 \cdot lb_j, \\ off^2 &= d_0 + d_1 \cdot lb_i + d_2 \cdot lb_j \end{aligned}$$

となる。

二重ループ中の配列操作では、二つのループ変数により配列データの操作順序が規定される。各ループのイタレーションの実行毎に発生する添字式値の変化はループ変数の増分値と、添字式の中でそのループ変数に乗算されている係数との積により計算される。我々は、このイタレーション毎の添字式値の変化を速度と呼ぶ。内側ループjにおける速度は一次元、二次元でそれぞれ

$$\begin{aligned} vel_j^1 &= c_2 \cdot st_j, \\ vel_j^2 &= d_2 \cdot st_j \end{aligned}$$

である。外側ループiについても同様にして、

$$\begin{aligned} vel_i^1 &= c_1 \cdot st_i, \\ vel_i^2 &= d_1 \cdot st_i \end{aligned}$$

と表現できる。

上記の初期値と速度を用いて配列添字式を書き改めると、一次元、二次元の配列添字式はそれぞれ

$$\begin{aligned} loc^1(i', j') &= off^1 + vel_i^1 \cdot i' + vel_j^1 \cdot j', \\ loc^2(i', j') &= off^2 + vel_i^2 \cdot i' + vel_j^2 \cdot j' \end{aligned}$$

となる。ここで、i'、j' はそれぞれ

$$0 \le i' \le \frac{ub_i - lb_i}{st_i},$$

$$0 \le j' \le \frac{ub_j - lb_j}{st_j}$$

の範囲を増分1で遷移する。配列添字式を正規化することで、タスク間の配列操作順序はループ変数の振舞いに関わらず、初期値と速度のみで表現可能となり、操作順序の類似性もこれらを用いて解析できる。

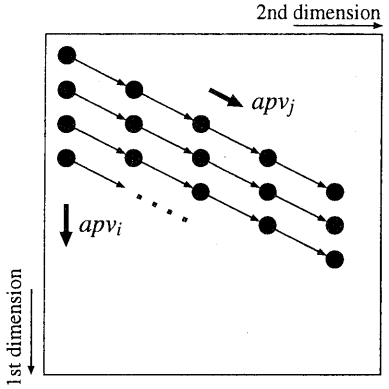

図 4: 配列データ操作順序の摸式図

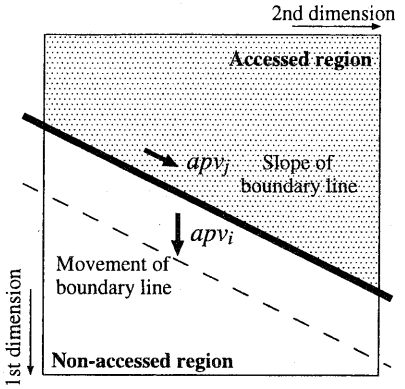

図 5: アクセス・パターン

3.3 アクセス・パターンの表現法

我々は、各次元における速度をアクセス・パターンとして定義し、以下のようなアクセス・パターン・ベクトルで表現する。

$$\begin{aligned} \overrightarrow{apv_j} &= (vel_j^1, vel_j^2) \\ &= (c_2 \cdot st_j, d_2 \cdot st_j), \\ \overrightarrow{apv_i} &= (vel_i^1, vel_i^2) \\ &= (c_1 \cdot st_i, d_1 \cdot st_i) \end{aligned}$$

ここで、内側ループのアクセス・パターン・ベクトルが $\overrightarrow{apv_j}$、外側ループのアクセス・パターン・ベクトルが $\overrightarrow{apv_i}$ であるときの二次元配列上での配列操作順序を摸式的に図示すると、図4のようになる。図中の正方形は配列 A に関する二次元配列空間で、左上が A[0][0] を表す。縦軸は鉛直下向きに一次元目の添字式値 (例えば、下向きに A[0][0], A[1][0], A[2][0], ...) を、また横軸は右向きに二次元目の添字式値 (例えば、右向きに A[0][0], A[0][1], A[0][2], ...) を表している。また、黒丸は操作される要素を表し、要素間を繋ぐ矢印は要素の操作順序を表している。この例では、まず外側ループの最初のイタレーションで A[0][0] から内側ループのアクセス・パターン・ベクトル $\overrightarrow{apv_j}$ の方向へ操作箇所が遷移する。内側ループの実行が終了すると、外側ループのループ変数の変化により $\overrightarrow{apv_i}$ にずれた後、再び $\overrightarrow{apv_j}$ の方向に操作していく。

ここで、内側ループで操作される要素、つまり $\overrightarrow{apv_j}$ の傾きを持つ直線上に配置された要素を集合として考える。この要素集合は外側ループ・イタレーションの実行とともに $\overrightarrow{apv_i}$ の方向に遷移する。したがって、ループ文実行途中のある時点において内側ループで操作された要素集合 (傾き $\overrightarrow{apv_j}$ の直線) に対して $\overrightarrow{apv_i}$ の方向はこれから操作する可能性のある領域で、$\overrightarrow{apv_i}$ の反対方向は今後操作する可能性がない領域であると捉えることができる。つまり、配列データの操作状況は図5のように表すことができる。二次元配列空間内で網掛けされた部分は、今後のループ文の実行で操作されない領域 (accessed region、以後、既操作領域と呼ぶ) で、白塗り部分は今後操作される可能性のある領域 (non-accessed region、以後、未操作領域と呼ぶ) を表している。これらの二つの領域は、図5の太線で表される傾き $\overrightarrow{apv_j}$ の境界線により分割される。ループ文の実行により操作される配列データは増加するが、この操作状況は境界線の移動により表現できる。また、外側ループのイタレーション毎に発生する境界線の移動方向・距離は、$\overrightarrow{apv_i}$ で表現でき、これを境界線の移動速度と呼ぶ。

このように、二本のアクセス・パターン・ベクトルにより表現される境界線の傾き及び移動速度は、二次元配列操作の規則性を特徴付ける重要な要素である。それぞれのタスクの境界線とその移動方向の違いが、漸進処理適用により実現される並列性に影響を与える。

4 アクセス・パターン類似性評価

本章では、3.3節で述べたアクセス・パターンの類似性の評価法について述べる。本章では、境界線の傾き・移動速度の差異が漸進処理適用時にどのような影響を与えるかを考察する。4.1節で境界線の傾きの違いから生じる参照タスクの実行開始時期の遅延を考察し、4.2節では境界線の移動速度の違いがタスク間の並列実行に与える影響を考察する。そして、4.3節で、アクセス・パターン類似度の計算手法を述べる。

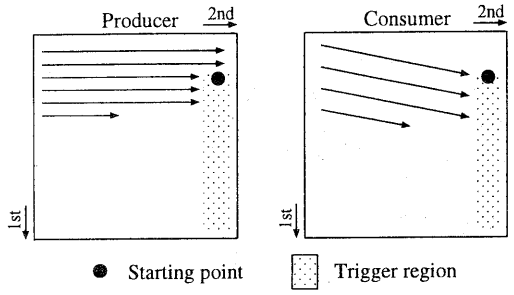

図 6: 境界線の傾きの異なるタスクの例

4.1 参照タスクの実行開始時期

アクセス・パターン間で境界線の傾きの違いが配列操作の並列実行に与える影響について考察する。図6に示すように、初期値が同一であるが、境界線の傾きの異なる配列操作を例に挙げて説明する。図中の矢印は、内側ループのアクセス・パターン・ベクトルの方向、つまり既・未操作領域間の境界線を意味する。

参照タスクの最初のイタレーションで操作される要素の中で、更新タスクにおいて最後に更新されるのは図中の黒丸で示される要素である。したがって、この要素が更新タスクで操作されるまで参照タスクの最初のイタレーションは実行できない。言い換えると、更新タスクがこの要素を操作するまでの間は、並列実行は行われない。我々は、この要素を参照開始点 (starting point) と呼び、更新タスクが参照開始点を操作するまでに必要な外側ループのイタレーション回数を t_p^{seq} と表現する。t_p^{seq} が小さいほど、効率のよい並列実行を期待できる。

4.2 イタレーション実行比

前節で述べたように、更新タスクで t_p^{seq} 回のイタレーションが実行された後、参照タスクの実行を開始できる。それ以後の配列操作間の並列実行性は、境界線の移動速度、つまり外側ループのアクセス・パターン・ベクトルに依存する。以後、更新・参照タスクの境界線の移動速度をそれぞれ $\overrightarrow{apv_p}$, $\overrightarrow{apv_c}$ と表現する。

参照開始点を $\overrightarrow{apv_c}$ だけ移動させた要素は、参照タスクの二番目のイタレーションを実行させるために必要なデータの中で、更新タスクで最後に操作される要素である。つまり、参照タスクの外側ループの各イタレーションを実行開始するための条件は、始点が参照開始点で傾き $\overrightarrow{apv_c}$ の線分上に存在する要素が更新タスクで操作済みであることである。我々はこの直線上に配置された要素集合をトリガ領域 (trigger region) と呼ぶ。

トリガ領域上での境界線の移動速度が二つのタスク間の並列実行性に与える影響として、イタレーション実行比が挙げられる。イタレーション実行比とは、更新タスクである一定回数のイタレーションが実行される度に、参照タスクで実行可能なイタレーション数の割合を比で表したものである。例えば、更新タスクと参照タスクのイタレーション実行比が 2 : 1 の場合は、更新タスクの2回のイタレーションが実行される度に参照タスクの1回のイタレーションが実行可能であることを意味する。イタレーション実行比は、トリガ領域上での各タスクの境界線の移動速度から計算可能である。トリガ領域上での境界線移動速度の大きさをそれぞれ $|\overrightarrow{apv_p}|$, $|\overrightarrow{apv_c}|$ と表現するイタレーション実行比は、

$$\text{更新タスク} : \text{参照タスク} = |\overrightarrow{apv_c}| : |\overrightarrow{apv_p}|$$

となる。

4.3 アクセス・パターン類似度

前節までで述べたように、参照タスクの実行開始時期及びイタレーション実行比は、アクセス・パターンの類似性や漸進処理適用時の実行効率を評価する上で重要な指標である。我々は、アクセス・パターン類似度を次に示す二種類の類似度の積により定義する。

4.1節で述べたように、境界線の傾きの違いは参照タスクの実行開始時期に影響を与える。t_p^{seq} が0、すなわち更新タスクの最初のイタレーションで参照開始点が操作される場合は、参照操作開始までの実行中断状態は発生せず、高い並列実行性の抽出が期待できる。逆に t_p^{seq} が更新タスクの全イタレーション回数と等しい場合は、図1と同様に配列操作間の並列実行性は全く抽出できない。つまり、参照タスクの実行開始時期の観点から、更新タスクの全イタレーション数に占める t_p^{seq} の割合が小さい場合はアクセス・パターン類似度は高く、また t_p^{seq} の割合が高ければ類似度は低く評価される。したがって、境界線の傾きの違いから生じるアクセス・パターン類似度 Sim_{slope} を次式により計算する。

$$Sim_{slope} = \frac{t_p^{all} - t_p^{seq}}{t_p^{all}}$$

次に、4.2節で述べたイタレーション実行比の観点からアクセス・パターン類似度を規定する。イタレーション実行比は、タスク間で配列操作の進行具合を規定する要素であり、1 : 1 になる場合二つのタスクの配列操作は同じような速度で進行することが期待でき、最も類似したアクセス・パターンであると評価できる。逆に、実行比がどちらか一方に偏っている場合は、一方のタスクの配列操作が進行しても、他方の配列操作があまり進行しない状況

が発生するため、アクセス・パターン類似度は低いと判断できる。したがって、我々はイタレーション実行比の違いから生じるアクセス・パターン類似度 $Sim_{velocity}$ を次式のように定義する。

$$Sim_{velocity} = \begin{cases} \dfrac{|\overrightarrow{apv_c}|}{|\overrightarrow{apv_p}|} & (|\overrightarrow{apv_c}| \leq |\overrightarrow{apv_p}|) \\ \dfrac{|\overrightarrow{apv_p}|}{|\overrightarrow{apv_c}|} & (|\overrightarrow{apv_c}| > |\overrightarrow{apv_p}|) \end{cases}$$

以上のことから、我々は参照タスクの実行開始時期やイタレーション実行比の影響を考慮したアクセス・パターン類似度 $Similarity$ を次に示す評価式で定義する。

$$Similarity = Sim_{slope} \cdot Sim_{velocity}$$

個々のタスクのイタレーションの実行時間や回数がほぼ同一であると仮定すると、上記評価式により得られる類似度は、漸進処理を適用した場合にタスク間でオーバラップされる配列操作の割合を表現している。

5 おわりに

本稿では漸進処理適用時のプログラム解析として、二次元の配列操作を対象としたアクセス・パターン解析手法を提案した。各ループでのイタレーション実行後の変化をアクセス・パターン・ベクトルとして定義し、これらをループ文実行途中で既・未操作領域を分割する境界線とその移動速度として捉えることでアクセス・パターンを表現した。また、境界線の傾きの違いから生じる参照タスクの実行開始時期の遅延や境界線の移動速度の違いから生じるイタレーション実行比を考慮したアクセス・パターン類似度の評価式を提案した。この評価式により得られるアクセス・パターン類似度は、漸進処理適用時の配列操作の実行効率を推定するための指標として利用できる。

今後の課題として、今回定義した類似度評価式の妥当性の確認とともに、より複雑な配列操作への対処が挙げられる。多次元配列を対象とした場合はアクセス・パターン・ベクトルの次元や本数が増加するため、今回のように境界線とその速度を用いて二次元平面上でアクセス・パターンを捉えることは不可能である。したがって、多重ループ中の多次元配列操作を考慮したアクセス・パターン解析手法を検討していく必要がある。また、添字式がループ変数の線形結合で表現されない配列操作への対処も今後検討していく予定である。

謝辞

日頃からご鞭撻、ご教授いただく中京大学情報科学研究科・福村晃夫教授、本学工学研究科・稲垣康善教授、鳥脇純一郎教授に深謝するとともに、日々討論いただく研究室の皆様に感謝します。

参考文献

[1] H.Zima and B.Chapman: "*Supercompilers for Parallel and Vector Computers*", Addison-Wesley Publishing Company (1991).

[2] M.Wolfe: "*High Performance Compilers for Parallel Computing*", Addison-Wesley Publishing Company (1996).

[3] D.Padua and M.Wolfe: "Advanced Compiler Optimizations for Supercompilers", *Communications of the ACM*, Vol. 29, No. 12, pp. 1184–1201 (1986).

[4] 朝倉宏一, 渡邉豊英: "ワークステーション・クラスタ環境における並列化コンパイラの構成", 電気学会論文誌 C, Vol. 118-C, No. 4, pp. 558–568 (1998).

[5] 三田勝史, 朝倉宏一, 渡邉豊英: "配列データ分割による漸進的並列処理手法の適用", 並列処理シンポジウム JSPP '98, p. 147 (1998).

[6] Y.Paek, J.Hoeflinger and D.Padua: "Access Regions: Toward a Powerful Parallelizing Compiler", *Technical report in Univ. of Illinois at Urbana-Champaign*, Center for Supercomputing Res. & Dev. (1996).

[7] J.Hoeflinger, Y.Paek and D.Padua: "Region-based Parallelization Using the Region Test", *Technical report in Univ. of Illinois at Urbana-Champaign*, Center for Supercomputing Res. & Dev. (1996).