卒業論文

DES 暗号の並列化の検討

氏　　名：中西 純也
学籍番号：22600602071-4
指導教員：山崎 勝弘　教授
提 出 日：2010 年 02 月 19 日

立命館大学理工学部電子情報デザイン学科

内容梗概

並列処理には大規模な問題でも計算時間を大幅に短縮できるというメリットがある。単一プロセッサの速度限界が近づく中で欠かせない技術の一つである。近年では、共有メモリ計算機の普及に伴い、並列プログラミングも分散メモリ環境から、共有メモリ環境へと移行しつつある。その共有メモリ用のプログラミングモデルとして現在注目を集めているのが、OpenMP である。OpenMP は移植性が高く、プログラミングも比較的簡単なので、今後並列プログラミングの主流になると期待されている。また、高性能な PC が安価で手に入るようになり、高速なネットワーク環境が普及してきた事から高性能な PC クラスタの構築が可能となった。

本論文では、ECB モードを使った DES 暗号の並列化を述べる。また、ECB モード以外の処理モードについての考察も述べる。

目次

図目次

表目次

1. はじめに

コンピュータにおいて複数のプロセッサで 1 つのタスクを動作させる事を並列処理と言う。PC クラスタを用いた並列処理を行うと、大規模な計算の大幅な時間短縮が可能となる。PC クラスタとは複数台の PC を組み合わせ、ひとまとまりのシステムにしたものである。並列処理を行うには、複数台の PC をネットワークで接続し、さらに、その複数台の PC を一つのシステムのように扱えるようにする PC クラスタ専用のソフトウェアと、PC クラスタに計算させるための並列プログラムが必要である。これらの要件を満たして、PC クラスタは構成要素である複数の PC を一つのシステムのように扱うことが出来き、並列処理を行うことができる。

近年、PC クラスタの構成要素となる PC 自体の性能が近年大きく向上しているため、PC クラスタの性能は著しく向上している。例えば、今まででは計算に時間がかかりすぎて実現不可能と言われていた、地球規模での気象予測、DNA 解析などが、現在では PC クラスタで実現されている。PC クラスタ自体の性能も向上しており、時間がかかりすぎて不可能な計算、将来的には可能になるであろう。

このように PC クラスタは、主にプログラムの高速化のために利用され、主なメモリモデルとして次の３種類が存在する。プログラムは並列処理を行っている全ての PC のメモリにアクセスすることができ、物理メモリを共有して処理を行う SMP(共有メモリモデル)。プロセッサは他のプロセッサの主記憶の読み書きを行うことができる分散共有メモリモデル。プロセッサは他のプロセッサの主記憶を読み書きすることはできない分散メモリモデルの３種である。

また、並列プログラミングには大きく分けて２つあり、分散メモリプログラミングと共有メモリプログラミングである。分散メモリプログラミングにはメッセージパッシング式の MPI や PVM、共有メモリプログラミングには fork-join 式の OpenMP を用いる事ができる。

本研究では、DES 暗号を対象に、処理モードの一つである ECB モードの並列処理を行って、暗号化の処理時間の向上を図っている。本研究での DES 暗号の並列化には、本研究室が所持する Nycto クラスタを使用する。

現在、代表的な現代暗号方式は秘密鍵暗号と公開鍵暗号に別れており、秘密鍵暗号方式は家の鍵と同じように、閉める（暗号化）鍵と開ける（復号）鍵が同じで、個人が秘密に鍵を管理する方式である。代表的な暗号には DES 暗号や FEAL 暗号があり、秘密鍵暗号方式の標準となっているのは DES(Data Encryption Standard)である。暗号と復号鍵が共通なため、通信相手とお互いに鍵を共有することになり、通信相手の数だけの鍵を必要とする。

一方、公開鍵暗号方式はウェブ上などで公開された暗号鍵により誰でも暗号化でき、正規のものだけが持つ秘密鍵により復号出来る方式である。代表的な暗号には RSA 暗号(RSA は開発者 3 人の頭文字)や楕円曲線暗号がある。公開鍵暗号として標準となってるのは RSA

暗号である。暗号鍵はネット上などで公開しておくので、自分の秘密鍵のみを秘密に保有していればよい。

公開鍵暗号では暗号化と復号の順序を逆にすると、秘密鍵を保有するものだけが秘密変換した結果を、その人の公開鍵で復号し、個人を特定することができる。この認証機能は秘密鍵方式では困難である。また、秘密鍵方式のアルゴリズムはデータの並べ替えや置き換えによるもので、多精度の剰余演算を必要とする公開鍵暗号方式に比べて、処理速度は高速である。そのため、データの暗号化には共通鍵暗号方式が用いられ、認証やディジタル署名および秘密鍵暗号の鍵配送などには公開鍵暗号を用いたハイブリッド方式を採用しているのが一般的である。

本論文では、本章で研究全体の背景と目的を明らかにし、第 2 章で DES 暗号のアルゴリズムと並列化を述べる。第 3 章では、実験環境について述べ、第 4 章では並列化の実験と考察について述べる。

2. DES 暗号のアルゴリズムと並列化

2.1 現代暗号(秘密鍵暗号)

現代暗号は暗号鍵と復号鍵が同じである秘密鍵（対称）暗号と、暗号鍵を公開し、復号鍵を秘密に保持する公開鍵（非対称）暗号に大別される。暗号のアルゴリズムを長い間秘密にしておくことは困難であることから、その安全性はアルゴリズムの秘密性に依存してはならず、鍵の秘密性に基づかなければならない。アルゴリズムを公開することにより、研究開発が活性化し、その進化を促すことになる。秘密鍵方式で標準となっている DES 暗号についてのアルゴリズムを解説する。

2.2 DES 暗号のアルゴリズム

DES 暗号は 64 ビットの平文を 64 ビットの鍵で暗号化し、同じ 64 ビットの鍵で復号する 64 ビットのブロック暗号である。主な処理はビットの順序を配列し直す転置、あるビット列を別のビット列に置き換える換字および排他的論理和演算である。転置にはブロックの入出力数が変わらない単純転置、ブロックの入力数よりも出力数が大きくなる拡大転置およびブロックの入力数よりも出力数が小さくなる縮小転置がある。換字は s ボックスという 8 個の換字表に従って、それぞれが 6 ビットを 4 ビットに置き換える。その結果、48 ビットを 32 ビットに換字することになる。DES 暗号の処理アルゴリズムを図 1 に示す。

図 1 で示すように、共通鍵から導き出された副鍵 K1～K16 で 16 段階（ラウンド）の暗号関数（f）による処理を繰り返して暗号化・復号を行う。暗号化と復号は同じアルゴリズムであるが、副鍵の使用順序が逆になる。暗号化の処理手順を次に説明する。

64 ビットの平文は初期転置（Initial Permutation:IP）により単純転置を施され、上位 32 ビットはレジスタ L_0 に、下位 32 ビットはレジスタ R_0 にそれぞれ格納される。

ラウンド 1 では R_0 と K1 に対する暗号関数 f の結果と L_0 との排他的論理和をとって R_1 とする。一方、R_0 を L_1 に格納する。暗号関数 f については後で説明する。

$$L_1 = R_0$$

$$R_1 = L_0 \oplus f(R_0, K_1)$$

ここで、⊕はビット列の排他的論理和を意味する。図では省略してあるが、平文の 32 ビットは 48 ビットに拡大転置され、48 ビット同士の排他的論理和が行われることになる。このような処理を 16 ラウンド繰り返す。各ラウンドの関係式を以下に示す。

$$L_r = R_{r-1}$$

$$R_r = L_{r-1} \oplus f(R_{r-1}, Kr) \quad \text{ここで、} 1 \le r \le 16$$

16 ラウンドの結果、得られた L_{16} と R_{16} を逆にして逆初期転置（IP^{-1}）を施し、64 ビットの暗号文を得る。

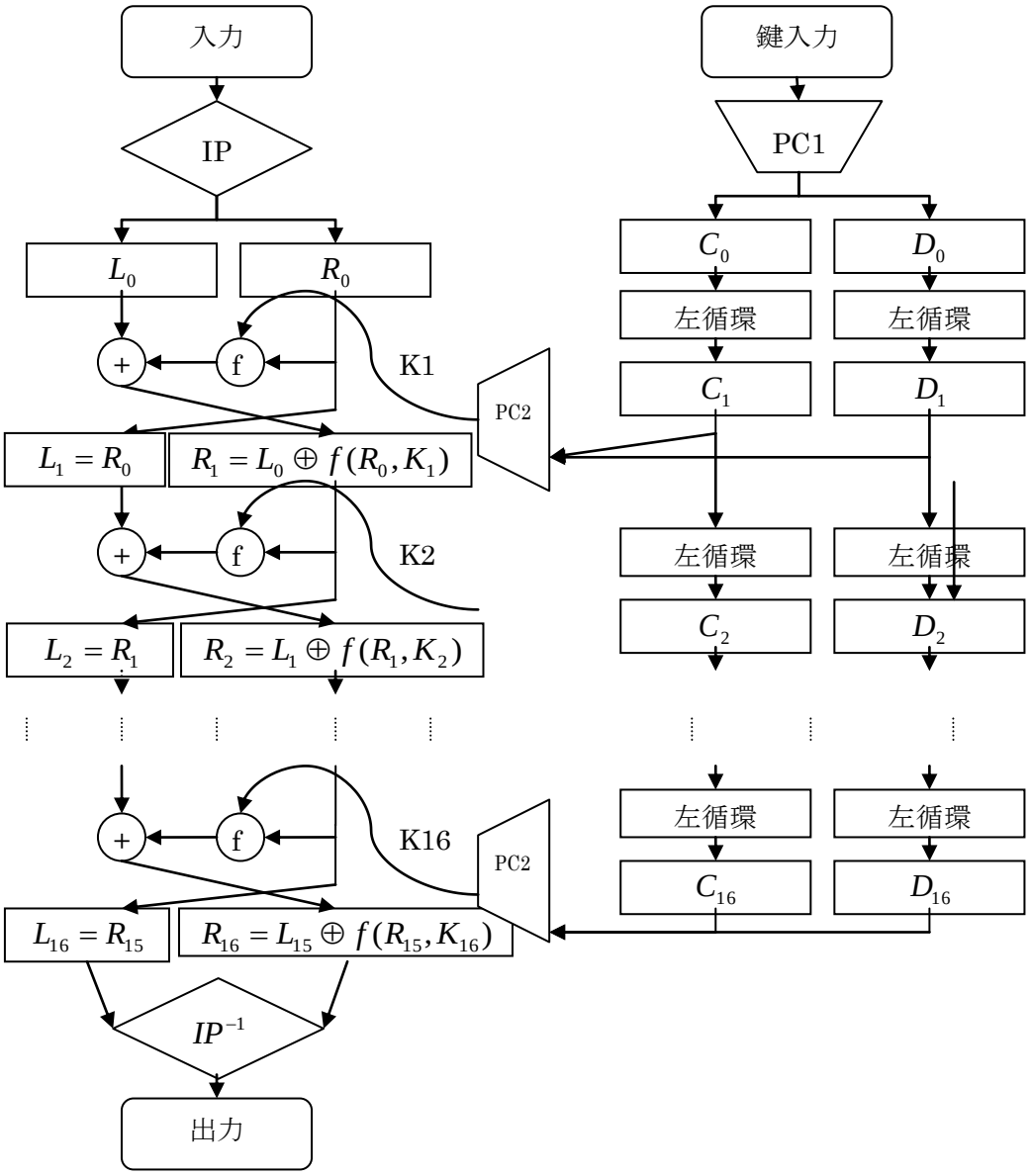


図１：DES 暗号の処理アルゴリズム

（１）初期転置と逆初期転置

次に、初期転置（IP）と逆初期転置（IP^{-1}）について説明する。64 ビットの入力がM_1～M_{64}、転置後の出力をT_1～T_{64}として図で表すと以下のようになる。

入力　M_1・・M_7・・・・M_{58}・・M_{64}

出力　T_1・・・・・・・・・・・・T_{64}

IP と IP^{-1} は相互関係から逆関数になっていることがわかる。

（2）副鍵の生成

一方、64 ビットの鍵は第 8,16,24,32,40,48,56,64 番目のビットパリティチェック用に用いられる。暗号化・復号処理過程ではパリティビットを除いた 56 ビットに対して、PC1 で縮小転置される。64 ビットの鍵入力の内、8, 16,24,32,40,48,56,64 ビットが出力されないことにより、縮小転置になっている。

PC 1 により縮小転置を施して出力された 56 ビットは、28 ビットに 2 分割されてレジスタ C と D に格納される。各レジスタ内 28 ビットのデータにはそれぞれ別々にラウンド 1,2,9 および 16 では 1 ビットの左回転シフト、それ以外のラウンドでは 2 ビットの左回転シフトが行われる。レジスタ C と D の計 56 ビットは PC2 によって 48 ビットに縮小転置され、それぞれのラウンド用に副鍵 K1 から順に K16 を生成する。転置前の 9,18,22,25,35,38,43,54 が欠番となっており、56 ビットが 48 ビットに縮小される。

（3）暗号関数 f

図 1 に示した 16 ラウンドの処理は、前段からの入力（L_{r-1}, R_{r-1}）に暗号関数 f を施して、次段に(L_r, R_r)を出力する。その関係は前述したように、

$$L_r = R_{r-1}$$

$$R_r = L_{r-1} \oplus f(R_{r-1}, Kr) \quad \text{ここで、} 1 \le r \le 16$$

であり、排他的論理和、換字および転置からなる。暗号関数 f の処理アルゴリズムを図 2 に示す。

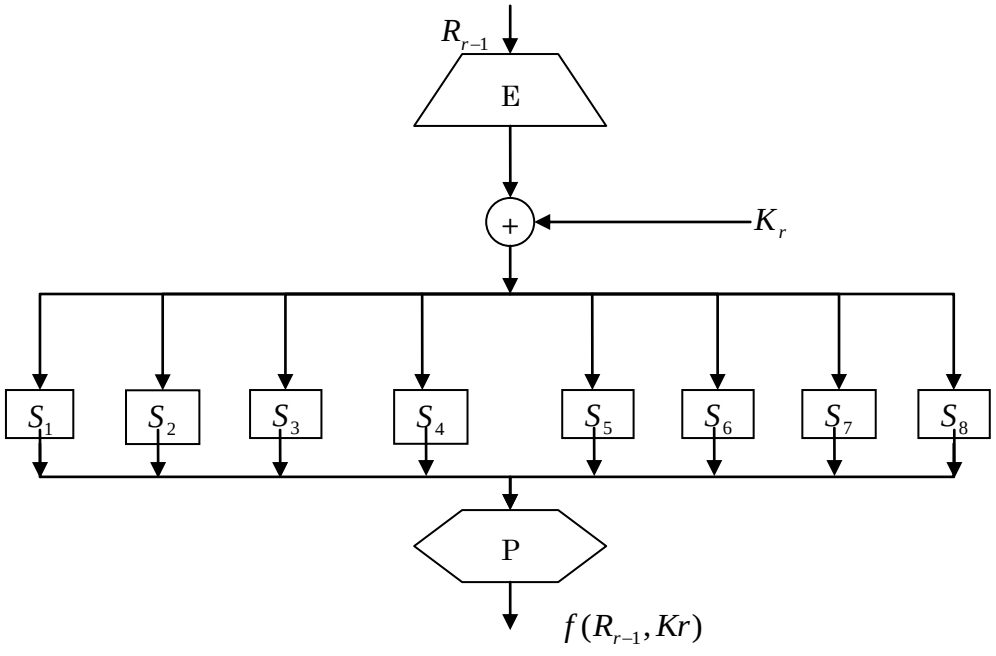


図 2 ：暗号関数 f の処理アルゴリズム

32 ビットの R_{r-1} は転置 E により 48 ビットに拡大転置され、副鍵 K_r の 48 ビットとの排他的論理和演算を施される（転置前で 16 ビット位置が重複）。その出力は 6 ビットずつ 8 ブロックに分割され、S_1〜S_8 の S ボックスによりそれぞれ 4 ビットに換字が行われる。計 32 ビットの換字出力は P によって単純転置を施された結果が $f(R_{r-1}, Kr)$ である。更に、L_{r-1} と 32 ビットの排他的論理和演算を施し、ひとつのラウンドを終了する。

ここで利用する S ボックスを表 1 に示す。

表 1 ：S ボックス

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S2	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S3	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S4	0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S5	0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S6	0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S7	0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6

	2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S8	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

S ボックスによる換字手順を次に説明する。入力 6 ビットをb_5,b_4,b_3,b_2,b_1,b_0とする。MSB と LSB の"b_5b_0"に相当する 2 進数が表の行番号を指定し、中央の 4 ビット"b_4,b_3,b_2,b_1"が列番号を指定する。その行と列で指定された数値を 4 ビットの 2 進数に変換する。例えば、S_1の入力が"110100"である場合、行は"10"、列は"1010"であるので、2 行 10 列の$(9)_{10}$、従って、"1001"が変換結果として出力される。

2.3 逐次暗号処理過程

実際に、平文と鍵を設定して暗号化の処理を追っていく。それぞれを以下のように設定する。DES 暗号は 64 ビットのブロック暗号で、64 ビットごとに暗号処理される。簡単なために ASCII コードで 8 文字（64 ビット）の平文と鍵を設定する。

平文　⇒　It's ok?
平文の ASCII コード(16 進)　⇒　49　74　27　73　20　6F　6B　3F
鍵　⇒　DES KEY (DES と KEY の文字間と KEY の後にそれぞれスペースがある)

ASCII コード 7 ビットの LSB にパリティビットを付加して 8 ビットにする。7 ビットのコード中に 1 の数が偶数個あるいは奇数個になるようにパリティビットをセットする。それぞれ偶数パリティおよび奇数パリティといい、8 ビット中のエラー検出用に用いられている技法であるが、パリティチェックビットは鍵の処理では無視される。
例えば、"D"の ASCII コードは"1000100"であり"1"の数は 2 個で偶数である。偶数パリティとした場合、LSB に"0"を付加する。"10001000"で、"D"は 16 進数で"88"になり、鍵を 16 進数で表すと次のようになる。

鍵の 16 進数　⇒　88　8B　A6　41　96　8B　B2　41

以上のデータにおいて、暗号化におけるラウンド 1 をシミュレートしてそれぞれの処理を説明する。

（1）初期転置（IP）

平文”It’s ok?”の 8 文字 64 ビットに初期転置を施し、32 ビットに 2 分割して、レジスタ L と R にセットする。

レジスタ L ← 01101011 10001010 10100110 11101101

レジスタ R ← 00000000 11111110 11100001 11101100

（2）副鍵の生成

鍵”DES KEY ”の 64 ビットを 56 ビットに縮小転置（PC1）し、28 ビットに 2 分割して、レジスタ C と D にセットする。

レジスタ C ← 01110111 10001000 01000100 0101

レジスタ D ← 01110110 00010100 00100011 0000

ラウンド 1 では、レジスタ C と D にそれぞれ 1 ビットの左回転シフトを施す。

左回転シフトの結果、

レジスタ C ← 11101111 00010000 10001000 1010

レジスタ D ← 11101100 00101000 01000110 0000

レジスタ C と D の計 56 ビットを PC2 で縮小転置し、副鍵 K1 を生成する。

副鍵 K1 ← 01001110 01100010 01011001 10100010 10100010 10101010

（3）暗号関数 f

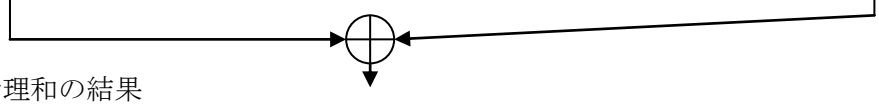
平文用のレジスタ R の 32 ビットを E で 48 ビットに拡大転置を施す。転置後の 48 ビットは生成された副鍵 K1 と排他的論理和演算が施される。その結果の 48 ビットは 6 ビットずつ 8 分割され S ボックスに与えられて、それぞれ換字表に従い 4 ビットに変換される。

縮小転置（PC2:K1）の出力

01001110 01100010 01011001 00100111 01100100 11000110

拡大転置（E）の出力

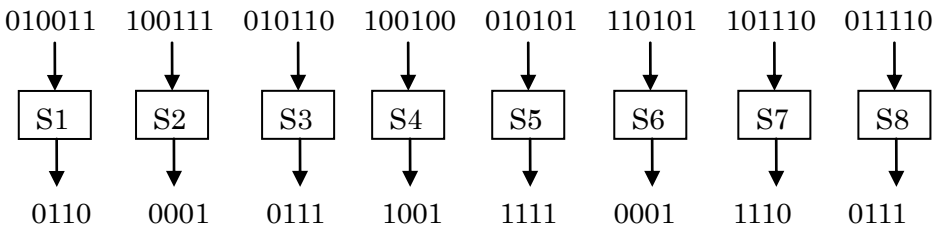
00000000 00010111 11111101 01110000 00111111 01011000



排他的論理和の結果

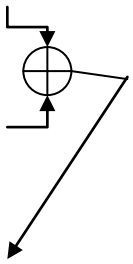
01001110 01110101 10100100 01010111 01011011 10011110

6 ビットずつ 8 分割して S ボックスへ



以上のようにして得られた 32 ビットの換字出力は P により単純転置される。

レジスタ L の出力　　01101011 10001010 10100110 11101101



単純転置（P）の出力　10100101 00010111 11101110 11100101

排他的論理和の結果　　11001110 10011101 01001000 00001000

この排他的論理和の結果は平文用のレジスタ R に格納され、ラウンド 1 を終了する。これをラウンド 16 まで繰り返し、逆初期転置を施して、暗号文を得る。

復号は前述したように暗号化と同じアルゴリズムであるが、副鍵の使用順序が逆になる。

2.4　DES 暗号の並列化手法

ここまで DES アルゴリズムの構成を説明してきたが、DES アルゴリズムは、データの安全性を保障するための基本的な構成要素の 1 つに過ぎない。DES をさまざまな状況で利用するために、5 つの処理モードが定義されている。これらの 5 つのモードは、実際上 DES が使われる状況のほとんどを網羅するように作られている。

本研究では 5 つの処理モードの中の 1 つである ECB モードを並列化している。ここでは、ECB モードの解説と、並列化について述べる。

（1）ECB モードの動作原理

ECB モードは平文長が n(n=64bit)の倍数であるような平文に対して暗号化を行う利用モードである。手法は平文を n ビット毎のブロックに分割し（それぞれを M_i とする）、それぞれ独立にブロック暗号の暗号化関数を入力とする。その結果えられた出力が暗号文ブロック（C_i）となり、暗号文はそれらを接続したものとなる。

$$C_i = Enc_K(M_i).$$

ここでの Enc_K とはあるブロック暗号の暗号化処理（鍵 K）を示す。この利用モードには初期値がなく、平文と鍵のみから生成される。復号化は上記の式の逆関数になる。各ブロックが独立する形式から、並列処理性と、数少ない処理順序不問性（Out-of-order）性がある。すなわち、ブロック単位でデータが入れ替わったとしても、その順序いれかえをすることなく、到着した順序に復号処理に渡すことができる特徴がある。当然、復号化結果は、到着順序に応じて並び替えなければ正しい平文には戻らない。ECB モードの処理アルゴリズムを図３に示す。

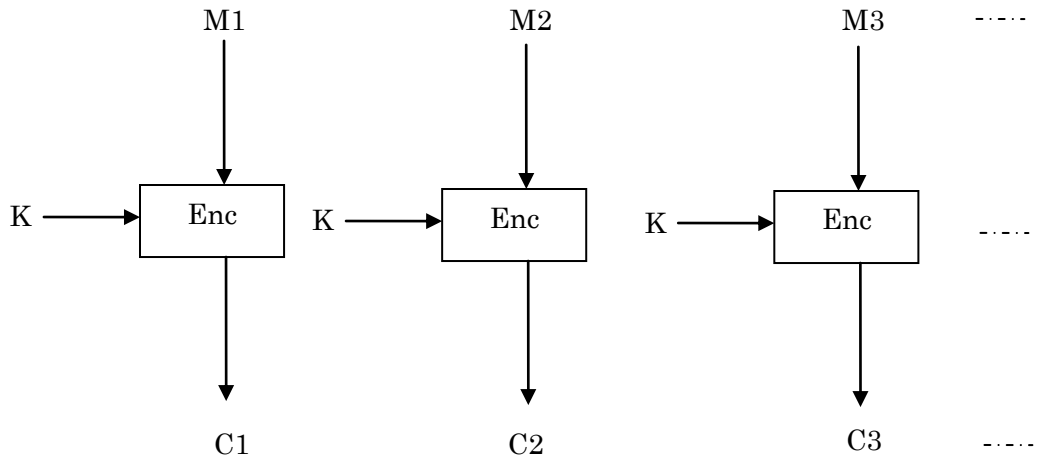


図３：ECB モードの処理アルゴリズム

上記のように、ECB モードは平文の各 64bit ブロックを、同じ鍵を用いて独立に暗号化するため、データの依存関係がない。

（２）ECB モードの並列化手順

本研究に使用している ECB モードでの暗号化は Enc であらわされる DES 暗号の処理を経て、平文から暗号文へと出力される。そのため、具体的には Enc の箇所を並列化する。入力された平文データは複数のブロックに分けられて処理されていくので、そのブロックを分割し、OpenMP による並列処理を行う。

並列処理においてデータの分割方法には、代表的なものとして block 分割と、cyclic 分割の２つの分割方法が存在する。本研究では block 分割を行って ECB モードの並列化を行う。ここで、OpenMP の説明を行う。

（３）OpenMP での並列化手法

OpenMP とは、共有メモリのマルチプロセッサ環境を利用するのに用いられる API（Application Programming Interface：アプリケーション・プログラミング・インターフ

ェース）であり、ベースとなる言語(Fortran, C/C++)に指示文(ディレクティブ)を追加することで並列プログラミングを行う。

OpenMP は fork-join モデルであり、複数のスレッド(プロセッサ)が並列プログラムを実行するとき、逐次部分は単一のスレッドで実行し、並列化された部分は他のスレッドを生成して並列処理を行い、並列指示文が終わると単一のスレッドのみに逐次実行に戻る。また、OpenMP は図 4 のように、プログラム中にあるノードの数により fork-join を繰り返す。なお、この図にある A～E は、あとに続く OpenMP の記述に対応している。

fork-join モデルとは、複数のスレッド（プロセッサ）が並列プログラムを実行するとき、逐次部分は単一のスレッド（マスタースレッド）で実行して、プログラムが並列指示文によって並列化された部分になると、ほかのスレッドを生成（fork）して並列処理を行う。並列指示文が終わると、単一スレッドのみの逐次実行（join）に戻る。このように、プログラム中にあるノードの数により fork-join を繰り返す。また、OpenMP において、逐次処理部を逐次リージョン（Sequential region）、並列処理部を並列リージョン（Parallel region）と呼ぶ。

C 言語で OpenMP を用いて並列処理を行うときは、並列化したい部分に次のような OpenMP の指示文を挿入する。

指示文：#pragma omp ～

このような指示文をプログラマが明示的に並列性を指示し挿入することによって、逐次プログラムから段階的・シームレスに並列化が可能となる。

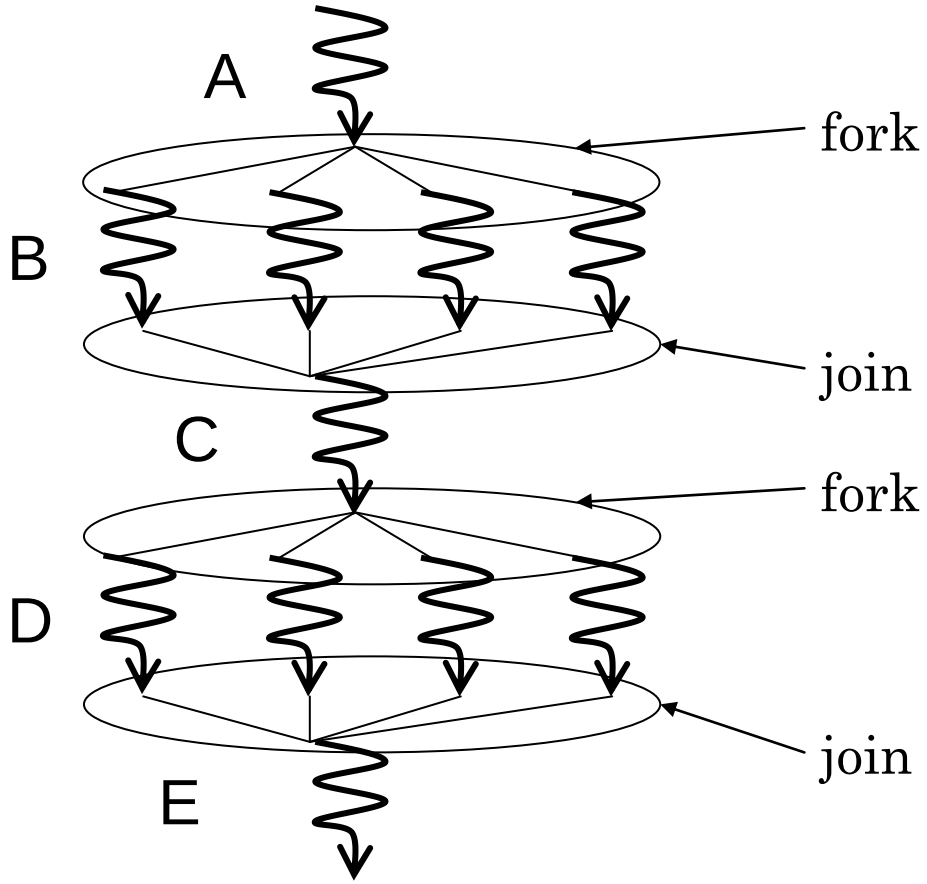


図４：OpenMP の fork-join モデル

図４の実際の記述モデルは以下のようになる。

```
#include <omp.h>

int main()
{
     A;
#pragma omp parallel
     {
        B;
     }
     C;
#pragma omp parallel
     D;
     E;
}
```

#pragma omp parallel の直後の文・ブロックは並列リージョンとなる。また、並列リージョンから呼ばれる関数も並列実行な特徴がある。

これより、ソースプログラムの ECB モード実行部分に#pragma を加え OpenMP に対応し、並列化を行える用に書き換え、Intel Compiler でコンパイルし、並列化を行った。

また、図５に ECB モードでのブロック処理モデルを示す。この図の M１～Mi は 64bit の平文ブロックを示し、C1～Ci は 64bit の暗号文ブロックを示す。このブロックの数は処理するデータの大きさによって異なる。K は鍵を示し、ECB モードなので共通の値である。この鍵の値と Mi の値を Enc の暗号化処理に従い暗号化を行い、Ci として出力する。この一連の流れを OpenMP により、並列化する。

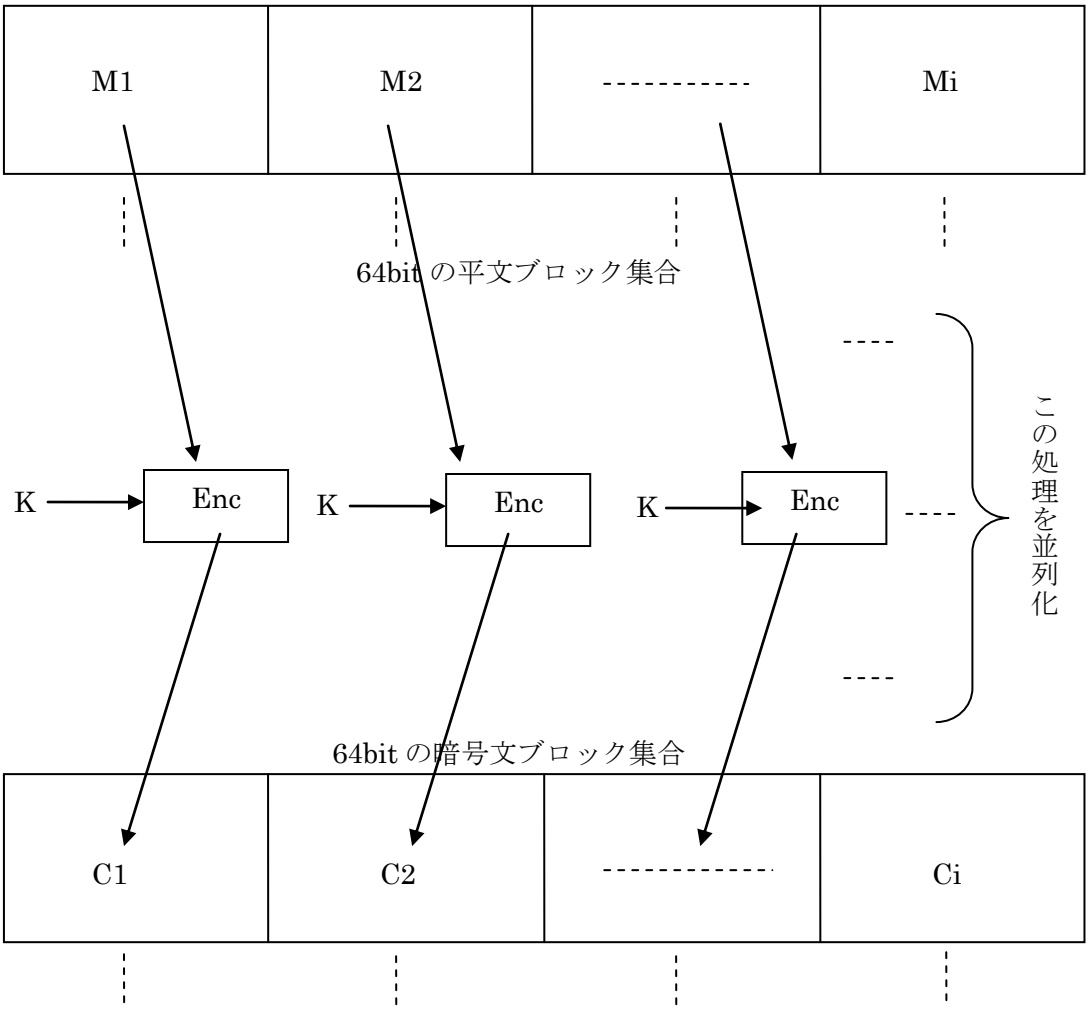


図５：ブロック処理モデル

3. 実験環境

実験に用いた Nycto クラスタの構成を図 6 に示す。Nycto クラスタは本研究室が所持するクラスタである。Nycto クラスタの計算ノードは、Nycto00 と Nycto01 の 2 台で、それぞれのノードに 2 個の Quad Xeon プロセッサが搭載されている。Nycto 一台のスペックは、Quad Xeon 3GHz × 2 台、8GB SDRAM、300GB HDD であり、これらは最大帯域幅 1Gbps の Gigabit Ethernet で接続されている。また、これらには Intel C/C++ Compiler がインストールされている。

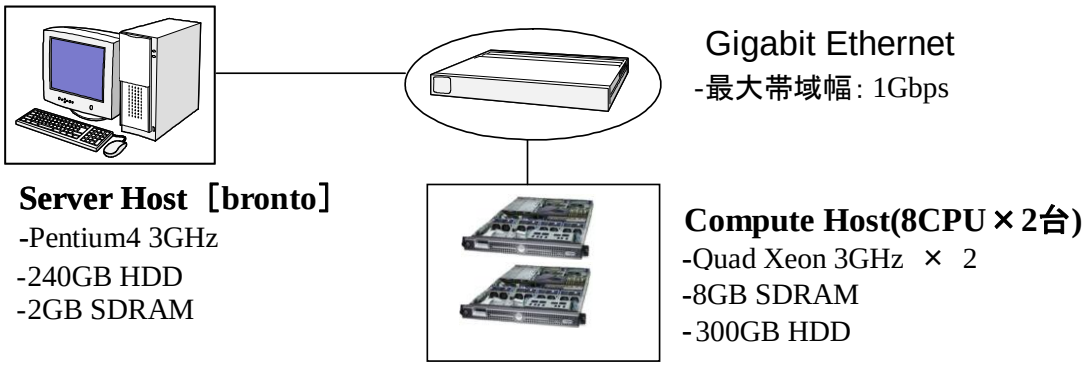


図 6：Nycto クラスタの構成

本研究では、マルチコア環境である Nycto 一台で、実行するプロセッサ数を変えて実験を行った。

4. 実験結果

4.1 ECB モードでの並列処理結果

使用するクラスタは Nycto で、処理するデータは小さいものから順に６４ビット、１２８ビット、２５６ビット、５１２ビット、１０２４ビット、２０４８ビット、４０９６ビットの７つのパターンに分けて処理を行った。ここでの N はブロック数で、64bit で 1 ブロックを示す。本研究の実験結果を表２に、速度向上比のグラフを図 7 に示す。

表２：ECB モードでの並列処理結果

N=1(64bit)

プロセッサ数	1	2	4	8
実行時間(秒)	0.8491			
速度向上比	1			

N=2(128bit)

プロセッサ数	1	2	4	8
実行時間(秒)	1.1048	0.9777		
速度向上比	1	1.13		

N=4(256bit)

プロセッサ数	1	2	4	8
実行時間(秒)	1.754	1.3703	0.6772	
速度向上比	1	1.28	2.59	

N=8(512bit)

プロセッサ数	1	2	4	8
実行時間(秒)	2.8487	1.9379	0.9592	0.4861
速度向上比	1	1.47	2.97	5.86

N=16(1024bit)

プロセッサ数	1	2	4	8
実行時間(秒)	5.0121	3.0939	1.5281	0.7699
速度向上比	1	1.62	3.28	6.51

N=32(2048bit)

プロセッサ数	1	2	4	8
実行時間(秒)	9.3301	4.9641	2.6207	1.331
速度向上比	1	1.88	3.56	7.01

N=64(4096bit)

プロセッサ数	1	2	4	8
実行時間(秒)	18.6326	9.3174	4.7141	2.4006
速度向上比	1	2	3.96	7.76

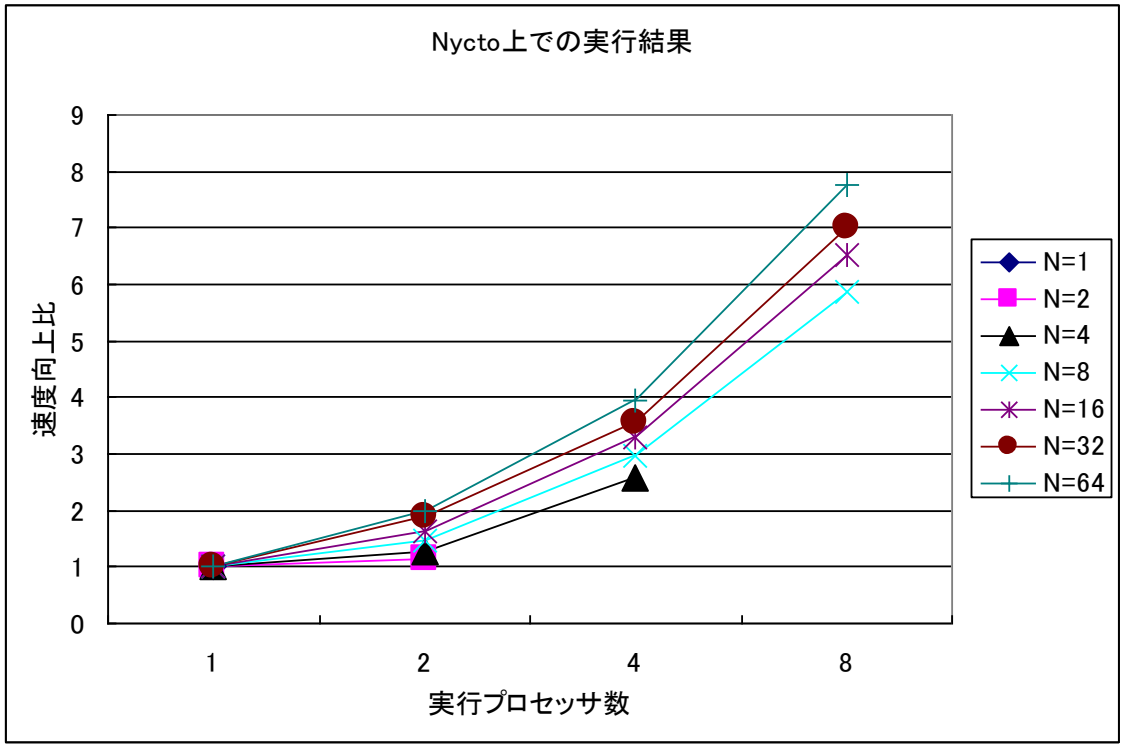


図 7：ECB モードの並列結果

4.2 考察

上記の結果から Nycto 上での実行結果は、計算量が増えるにしたがって理想的な速度向上比が得られるようになってくるということが言える。N の値（処理するビット数）が、大きくなればなるほど速度の向上が見られる。この高い速度向上比が得られた結果として、考えられる原因は ECB モードでのデータ処理が block 分割方法と相性が良いことであることがあげられる。block 分割は、通信回数自体が少なく、各プロセッサとの通信回数も 2 回ずつなので、ECB モードの並列化には適切なデータ分割方法であった。また、この傾向は、処理するデータが大きくなればなるほど、計算部分が大きいため、相対的に通信部分を占める割合を少なくすることができたと言える。

次に、ECB モード以外の処理モードと並列処理性について考察をする。

（1）CBC モード

次に処理する 64bit の平文と、前回の 64bit の暗号文の XOR を暗号化アルゴリズムの入力とする。ブロック単位の処理に一般的に利用され、前回の暗号文を利用するため、データの依存関係があり、暗号化では並列処理はできない。

（2）CFB（暗号フィードバック方式）モード

出力した暗号文のデータを送信する単位を j ビットであると仮定し、送信に必要な容量の無駄遣いを省いたモード。入力は j ビットずつ処理される。直前の暗号文を暗号化アルゴリズムへ入力し、擬似乱数ビット列を生成し、その結果と平文の XOR を取り、次の暗号文を作る。そのため、CBC モードと同様の理由から、暗号化では並列処理はできない。

（3）OFB（出力フィードバック方式）モード

上記のモードと似ているが、別の形である。暗号化アルゴリズムへの入力には直前の DES の出力を使う。こちらもデータの依存関係があり、並列処理はできない。

（4）CTR モード

CTR(カウンタ)モードは、初期値のみに依存し逐次的に擬似乱数を生成しながら暗号化を行う方法であり、任意のビット長の平文を処理できる性質がある。まず、平文を n(n=64bit) ビット毎のブロックに分割し（それぞれを M_i とする）、最後の端数の部分は端数ブロックとして扱う。開始値 SV を内部レジスタの初期値 R_1 とする。R_i をブロック暗号入力とし、暗号化処理の結果を H_i とする。これより暗号文ブロック $C_i = M_i \oplus H_i$ を生成する。次のブロックでは、内部レジスタ R を整数カウンタとして 1 数えあげる。また、Enc_K とはあるブロック暗号の暗号化処理（鍵 K）を示す。

$$C_i = M_i \oplus Enc_k(R_i),$$
$$R_{i+1} = R_i + 1.$$

ここでの開始値とは、特殊な運用が必要な初期値を指す。CTR が安全な処理モードであるために、同一の鍵が用いられている間は常に異なるブロック暗号入力を与える必要がある。CTR モードでは、内部状態の更新がカウンタであるため、システム要件から、カウンタの更新回数の限度などを知ることができる場合もある。このような情報を使いながら、上手く開始値を定義して、（同じ鍵のもとで）複数の平文を安全に暗号化できるようにする必要がある。図 8 に CTR モードの処理アルゴリズムを示す。

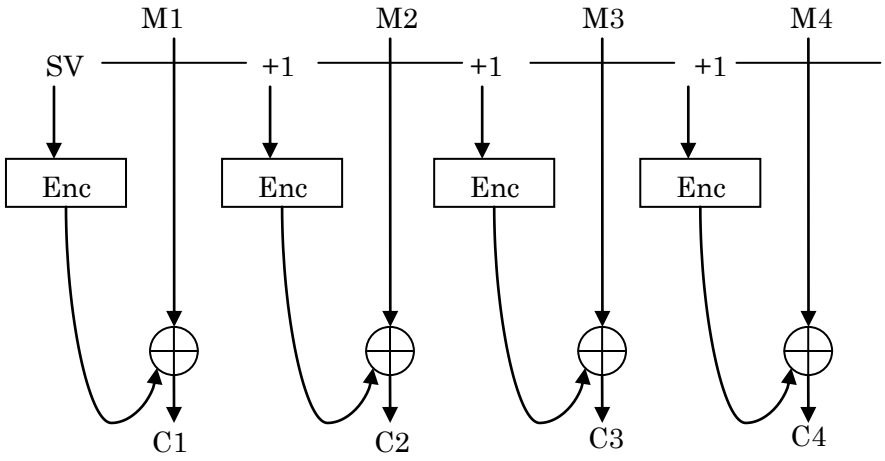


図８：CTR モードの処理アルゴリズム

暗号化復号化ともに並列処理性が実現可能であるが、処理しているブロックが平文の何ブロック目であるかという情報を処理系が知っている必要がある。従って、パイプライン化などのようなメカニズムで、メッセージを最初のブロックから処理する必要が出てくる。

同様に、何ブロック目のデータであるかがわかれば、処理順序不問性（Out-of-order）性も達成できる。すなわち、ブロック単位でデータが入れ替わったとしても、その順序いれかえをすることなく、到着した順序に復号処理に渡すことができる。当然、復号化結果は、到着順序に応じて並び替えなければ、正しい平文には戻らない。

5. おわりに

本研究では、PC クラスタ上で OpenMP を用いて、DES 暗号の ECB モードを対象として並列化を行った。実行台数を変えて、マルチコア CPU 環境で実行時間を測定し、実行プロセッサ数が８、処理するビットの大きさが４０９６ビットの時に、7.76 倍のほぼ理想的な速度向上が得られた。また、DES 暗号の並列化を行うことにより、並列プログラムの作成や、並列実行の仕方を確認することができた。 現在普及が進んでいるマルチコアの CPU では、並列処理は必要不可欠なものとなる。今後の研究課題として、今回、使ったクラスタである Nycto クラスタは本来 2 ノード１６プロセッサでの動作も可能である。しかし、実際の動作時には８プロセッサまでの実験しか不可であった。この検証が挙げられる。また、CTR モードのようなパイプライン化することにより、並列処理することができるモードの研究についての検証も挙げられる。

謝辞

本研究の機会を与えてくださり、数々の助言をいただきました山崎勝弘教授に心より感謝いたします。また、本研究にあたり、親切な指導や貴重なご意見をいただきました本研究室の皆様に心より感謝いたします。

参考文献

[1] ウィリアム・スターリングス：暗号とネットワークセキュリティ-理論と実際-，ピアソン・エデュケーション, 2001

[2] Douglas R. Stinson：暗号理論の基礎，共立出版, 1996

[3] 結城浩：暗号技術入門-秘密の国のアリス-，ソフトバンククリエイティブ, 2008

[4] 中村次男、笠原宏：パソコンで実習しながら学べる暗号のしくみと実装-プログラミングと演習-，日本理工出版会, 2009

[5] 赤間世紀：Java で学ぶ暗号プログラミング，秀和システム, 2003

[6] OpenMP：http://openmp.org/wp/

[7] インテル C/C++ コンパイラー OpenMP 活用ガイド
http://download.intel.co.jp/jp/business/japan/pdf/526J-002.pdf

[8] Omni OpenMP Compiler： http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/

[9] John L. Hennessy、David A. Patterson：コンピュータの構成と設計-ハードウェアとソフトウェアのインターフェース-（上）,日経 BP 社,1999

[10] John L. Hennessy、David A. Patterson：コンピュータの構成と設計-ハードウェアとソフトウェアのインターフェース-（下）,日経 BP 社,2006

付録：DES 暗号の並列化ソースコード

```
/*              des.c           Data Encryption Standard        */
#include <stdio.h>
#include <ctype.h>
#include <memory.h>
#include <omp.h>
/*#define               DEBUG       /* print out for debug */
#define         SC              1               /* = sizeof(char) */

typedef         unsigned int    UINT;

FILE *fp1, *fp2;
char buf[256], bufo[1024];
char *bufolast = bufo + 1016, *bufo_blen = bufo + 8;

UINT *key32[16], *key16[16];
UINT *message_IP[64];
UINT *key_L_s[16], *key_R_s[16], *temp_PC2M[48], *temp_PC2K[48];
UINT *mes_IP1[64], *mes_P1[32], *mes_P2[32], *mes_EK1[16], *mes_EK2[16];
UINT cipher[64], code[64], temp[64], mes[64];
UINT key_L[56], key_R[56];
UINT *des_in, *temp_28, *mes_32;

UINT *ini_len, *ini_len1, *xb1, *xb2;

static int mode = 1, count;
static int blen = 8, blen8 = 64, blen8a = 0, llen = 64, llen1 = 0;
static int blen8_M = 64 * sizeof(UINT), blen8a_M = 0;
static int llen_M = 64 * sizeof(UINT), llen1_M = 0;
static UINT initial[64], key[64];
static UINT x[64] =
        {1,1,1,1,1,1,1,1,   1,1,1,1,1,1,1,1,   1,1,1,1,1,1,1,1,   1,1,1,1,1,1,1,1,
         1,1,1,1,1,1,1,1,   1,1,1,1,1,1,1,1,   1,1,1,1,1,1,1,1,   1,1,1,1,1,1,1,1 };
static UINT IP[64] =
        {       57,49,41,33,25,17, 9,1,         59,51,43,35,27,19,11,3,
                61,53,45,37,29,21,13,5,         63,55,47,39,31,23,15,7,
```

```
                56,48,40,32,24,16, 8,0,          58,50,42,34,26,18,10,2,
                60,52,44,36,28,20,12,4,          62,54,46,38,30,22,14,6 };
static UINT IP1[64] =
        {       7,39,15,47,23,55,31,63,          6,38,14,46,22,54,30,62,
                5,37,13,45,21,53,29,61,          4,36,12,44,20,52,28,60,
                3,35,11,43,19,51,27,59,          2,34,10,42,18,50,26,58,
                1,33, 9,41,17,49,25,57,          0,32, 8,40,16,48,24,56 };
static UINT S[8][4][16] =
        {       {       {14, 4,13, 1,   2,15,11, 8,   3,10, 6,12,   5, 9, 0, 7 },
                        { 0,15, 7, 4, 14, 2,13, 1, 10, 6,12,11,   9, 5, 3, 8 },
                        { 4, 1,14, 8, 13, 6, 2,11, 15,12, 9, 7,   3,10, 5, 0 },
                        {15,12, 8, 2,   4, 9, 1, 7,   5,11, 3,14, 10, 0, 6,13 } },
                {       {15, 1, 8,14,   6,11, 3, 4,   9, 7, 2,13, 12, 0, 5,10 },
                        { 3,13, 4, 7, 15, 2, 8,14, 12, 0, 1,10,   6, 9,11, 5 },
                        { 0,14, 7,11, 10, 4,13, 1,   5, 8,12, 6,   9, 3, 2,15 },
                        {13, 8,10, 1,   3,15, 4, 2, 11, 6, 7,12,   0, 5,14, 9 } },
                {       {10, 0, 9,14,   6, 3,15, 5,   1,13,12, 7, 11, 4, 2, 8 },
                        {13, 7, 0, 9,   3, 4, 6,10,   2, 8, 5,14, 12,11,15, 1 },
                        {13, 6, 4, 9,   8,15, 3, 0, 11, 1, 2,12,   5,10,14, 7 },
                        { 1,10,13, 0,   6, 9, 8, 7,   4,15,14, 3, 11, 5, 2,12 } },
                {       { 7,13,14, 3,   0, 6, 9,10,   1, 2, 8, 5, 11,12, 4,15 },
                        {13, 8,11, 5,   6,15, 0, 3,   4, 7, 2,12,   1,10,14, 9 },
                        {10, 6, 9, 0, 12,11, 7,13, 15, 1, 3,14,   5, 2, 8, 4 },
                        { 3,15, 0, 6, 10, 1,13, 8,   9, 4, 5,11, 12, 7, 2,14 } },
                {       { 2,12, 4, 1,   7,10,11, 6,   8, 5, 3,15, 13, 0,14, 9 },
                        {14,11, 2,12,   4, 7,13, 1,   5, 0,15,10,   3, 9, 8, 6 },
                        { 4, 2, 1,11, 10,13, 7, 8, 15, 9,12, 5,   6, 3, 0,14 },
                        {11, 8,12, 7,   1,14, 2,13,   6,15, 0, 9, 10, 4, 5, 3 } },
                {       {12, 1,10,15,   9, 2, 6, 8,   0,13, 3, 4, 14, 7, 5,11 },
                        {10,15, 4, 2,   7,12, 9, 5,   6, 1,13,14,   0,11, 3, 8 },
                        { 9,14,15, 5,   2, 8,12, 3,   7, 0, 4,10,   1,13,11, 6 },
                        { 4, 3, 2,12,   9, 5,15,10, 11,14, 1, 7,   6, 0, 8,13 } },
                {       { 4,11, 2,14, 15, 0, 8,13,   3,12, 9, 7,   5,10, 6, 1 },
                        {13, 0,11, 7,   4, 9, 1,10, 14, 3, 5,12,   2,15, 8, 6 },
                        { 1, 4,11,13, 12, 3, 7,14, 10,15, 6, 8,   0, 5, 9, 2 },
                        { 6,11,13, 8,   1, 4,10, 7,   9, 5, 0,15, 14, 2, 3,12 } },
```

```
                {       {13, 2, 8, 4,   6,15,11, 1, 10, 9, 3,14,   5, 0,12, 7},
                        { 1,15,13, 8, 10, 3, 7, 4, 12, 5, 6,11,   0,14, 9, 2},
                        { 7,11, 4, 1,   9,12,14, 2,   0, 6,10,13, 15, 3, 5, 8},
                        { 2, 1,14, 7,   4,10, 8,13, 15,12, 9, 0,   3, 5, 6,11} } };
static UINT EK1[16] =
        {       63,36,35,40,39,44,43,48, 47,52,51,56,55,60,59,32 };
static UINT EK2[16] =
        {       31, 4, 3, 8, 7,12,11,16, 15,20,19,24,23,28,27, 0 };
static UINT PINV[32] =
        {        8,16,22,30, 12,27, 1,17, 23,15,29, 5, 25,19, 9, 0,
                 7,13,24, 2,   3,28,10,18, 31,11,21, 6,   4,26,14,20 };
static UINT PC1[56] =
        {       56,48,40,32,24,16, 8,    0,57,49,41,33,25,17,
                 9, 1,58,50,42,34,26,   18,10, 2,59,51,43,35,
                62,54,46,38,30,22,14,    6,61,53,45,37,29,21,
                13, 5,60,52,44,36,28,   20,12, 4,27,19,11, 3 };
static UINT PC2M[32] =
        {       16,10,23, 0, 27,14, 5,20, 18,11, 3,25,   6,26,19,12,
                51,30,36,46, 39,50,44,32, 48,38,55,33, 41,49,35,28 };
static UINT PC2K[16] =
        {       13, 4, 2, 9, 22, 7,15, 1, 40,54,29,47, 43,52,45,31 };
static UINT shift[16] =
        {       1, 2, 4, 6, 8, 10, 12, 14, 15, 17, 19, 21, 23, 25, 27, 28};
#ifdef  DEBUG
char kki[16] =
        {       '0', '1', '2', '3', '4', '5', '6', '7',
                '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
#endif
 /* prototype declaration */
void init_des(), makekey(), des(), usage(), openerr();
void hex_to_code(), byte_to_code(), code_to_byte();
void address(), xor(), ini_shift(), substitute();
void ecb(), cbc_e(), cbc_d(), ofb(), cfb_e(), cfb_d();
#ifdef          DEBUG
void bprint(), cprint(), code_to_hex();
```

```
void bprint(mes1, m, n)
UINT *mes1;
int m, n;
{
        UINT *p;
        p = mes1;
        do
        {
                printf("%d", *p++);
                if(--n == 0)
                {
                        printf(" ");
                        n = 8;
                }
        } while(--m);
}
void cprint(code1)
UINT *code1;
{
        UINT *p;
        int i;

        p = code1;
        i = 16;
        do
        {
                putchar(kki[(((*p++ * 2) + *p++) * 2 + *p++) * 2 + *p++]);
        } while(--i);
}
void code_to_hex(code1, hex)
UINT *code1;
char *hex;
{
        UINT *q;
        char *p;
        int i;
```

```
        p = hex;
        q = code1;
        i = 16;
        do
        {
                *p++ = kki[(((*q++ * 2) + *q++) * 2 + *q++) * 2 + *q++];
        } while(--i);
        *p = '¥0';
}
#endif
void hex_to_code(hex, code1)
char *hex;
UINT *code1;
{
        UINT *q = code1;
        char *p = hex;
        int ch, i = 16;
        while(ch = tolower(*p++))
        {
                if(ch >= '0' && ch <= '9')              ch -= '0';
                else if(ch >= 'a' && ch <= 'f')         ch = ch - 'a' + 10;
                else break;
                *q++ = (ch >> 3) & 1;
                *q++ = (ch >> 2) & 1;
                *q++ = (ch >> 1) & 1;
                *q++ = ch & 1;
                if(i-- == 1)            return;
        }
        memset(q, 0, i * 4 * sizeof(UINT));
}
void byte_to_code(byte, code1, cc)
char *byte;
UINT *code1;
int cc;
{
        UINT *q = code1;
```

```
        char *p = byte;
        int ch, n;
        n = (8 - cc) * 8 * sizeof(UINT);
        do
        {
                *q++ = ((ch = *p++) >> 7) & 1;
                *q++ = (ch >> 6) & 1;
                *q++ = (ch >> 5) & 1;
                *q++ = (ch >> 4) & 1;
                *q++ = (ch >> 3) & 1;
                *q++ = (ch >> 2) & 1;
                *q++ = (ch >> 1) & 1;
                *q++ =   ch          & 1;
        } while(--cc);
        memset(q, 0, n);
}
void code_to_byte(byte, code1)
char *byte;
UINT *code1;
{
        UINT *q = code1;
        char *p = byte;
        int n = blen;
        do
        {
                *p++                                                                    =
(((((((*q++*2)+*q++)*2+*q++)*2+*q++)*2+*q++)*2+*q++)*2+*q++)*2+*q++)*2+*q++;
        } while(--n);
}
void address(add, base, index, n)
UINT **add, *base, *index;
int n;
{
        UINT **pp = add;
        UINT *p = index;
        do
```

```
        {
                *pp++ = base + *p++;
        } while(--n);
}
void xor(bufout)
char *bufout;
{
        char *p = buf, *q = bufout;
        int n = count;
        do
        {
                *q++ ^= *p++;
        } while(--n);
}
void ini_shift()
{
        UINT *p = ini_len - 1, *q = initial + 63;
        int n = llen;
        while(n--)          *q-- = *p--;
        memset(initial, 0, llen1);
}
void substitute(code1, sub, n)
UINT *code1, **sub;
int n;
{
        UINT **pp = sub;
        UINT *p = code1;
        do
        {
                *p++ = *(*pp++);
        } while(--n);
}
void init_des()
{
        UINT **pp, **pp1, **qq;
        UINT *p, *q;
```

```
        int i;
        pp = key32;
        qq = key16;
        i = 16;
        do
        {
                *pp++ = (UINT *)malloc(32 * sizeof(UINT));
                *qq++ = (UINT *)malloc(16 * sizeof(UINT));
        } while(--i);
        des_in = (mode <= 2)? code: initial;
        temp_28 = temp + 28;
        mes_32 = mes + 32;
        address(key_L_s, key_L, shift, 16);
        address(key_R_s, key_R, shift, 16);
        address(temp_PC2M, temp, PC2M, 32);
        address(temp_PC2K, temp, PC2K, 16);
        address(mes_P1, mes,      PINV, 32);
        address(mes_P2, mes_32, PINV, 32);
        address(mes_EK1, mes, EK1, 16);
        address(mes_EK2, mes, EK2, 16);
        address(mes_IP1, mes, IP1, 64);
        address(message_IP, des_in, IP, 64);
}
void makekey(edflag)
int edflag;                                                    /*  1  :  encryption,  0  :
decryption */
{
        UINT **ps1 = key_L_s, **ps2 = key_R_s;
        UINT *p1, *p2, *q1, *q2;
        int i = 28, j, m = 28 * sizeof(UINT);
        p2 = (p1 = PC1) + 28;
        q1 = key_L;
        q2 = key_R;
        do
        {
                *q1++ = key[*p1++];
```

```
                *q2++ = key[*p2++];
        } while(--i);
        memcpy(key_L + 28, key_L, m);
        memcpy(key_R + 28, key_R, m);
        i = 15;
        do
        {
                memcpy(temp,       *ps1++, m);
                memcpy(temp_28, *ps2++, m);
                j = (edflag)? 15 - i: i;
                substitute(key32[j], temp_PC2M, 32);
                substitute(key16[j], temp_PC2K, 16);
        } while(--i >= 0);
}
void des(crypt, crypt_b)
UINT *crypt;
char *crypt_b;
{
        UINT **pp1, **pp2, **qq1 = key32, **qq2 = key16;
        UINT *p1, *p2, *q1;
        int i = 8, j, m;
        substitute(mes, message_IP, 64);
        do
        {
                pp1 = mes_P1;
                pp2 = mes_EK1;
                q1 = mes_32;
                p1 = *qq1++;
                p2 = *qq2++;
                j = 0;
                do
                {
                        m = S[j]
                                [(*(*pp2++)^ *p2++)*2 + *(*pp2++) ^ *p2++]
                                [(((*q1++ ^ *p1++)*2 + *q1++ ^ *p1++)*2 + *q1++ ^
*p1++)*2 + *q1++ ^ *p1++];
```

```
                        *(*pp1) ^= ((m >> 3) & 1);
                        pp1++;
                        *(*pp1) ^= ((m >> 2) & 1);
                        pp1++;
                        *(*pp1) ^= ((m >> 1) & 1);
                        pp1++;
                        *(*pp1) ^= (m & 1);
                        pp1++;
                } while(++j < 8);
                pp1 = mes_P2;
                pp2 = mes_EK2;
                q1 = mes;
                p1 = *qq1++;
                p2 = *qq2++;
                j = 0;
                do
                {
                        m = S[j]
                                [(*(*pp2++) ^ *p2++)* 2 + *(*pp2++) ^ *p2++]
                                [(((*q1++ ^ *p1++)* 2 + *q1++ ^ *p1++)* 2 + *q1++ ^
*p1++)* 2 + *q1++ ^ *p1++];
                        *(*pp1) ^= ((m >> 3) & 1);
                        pp1++;
                        *(*pp1) ^= ((m >> 2) & 1);
                        pp1++;
                        *(*pp1) ^= ((m >> 1) & 1);
                        pp1++;
                        *(*pp1) ^= (m & 1);
                        pp1++;
                } while(++j < 8);
        } while(--i);
        substitute(crypt, mes_IP1, 64);
        code_to_byte(crypt_b, crypt);
}
#pragma omp parallel
{
```

```
        void ecb()
        {
                //while(count = fread(buf, SC, blen, fp1))
                for(count=0; count = fread(buf, SC, blen, fp1); count++)
                {
                        byte_to_code(buf, code, count);
                        des(cipher, bufo);
                        fwrite(bufo, SC, blen, fp2);
                }
        }
}
void cbc_e()
{
        code_to_byte(bufo, initial);
        while(count = fread(buf, SC, blen, fp1))
        {
                xor(bufo);
                byte_to_code(bufo, code, blen);
                des(cipher, bufo);
                fwrite(bufo, SC, blen, fp2);
        }
}
void cbc_d()
{
        char *p = bufo, *q = bufo_blen;

        code_to_byte(bufo, initial);
        while(count = fread(q, SC, blen, fp1))
        {
                byte_to_code(q, code, count);
                des(cipher, buf);
                xor(p);
                fwrite(p, SC, blen, fp2);
                if(count != blen)   return;
                if((p = q) < bufolast)          q += blen;
                else
```

```
                {
                        memcpy(bufo, p, 8);
                        p = bufo;
                        q = bufo_blen;
                }
        }
}
void ofb()
{
        des(code, bufo);
        while(count = fread(buf, SC, blen, fp1))
        {
                xor(bufo);
                fwrite(bufo, SC, blen, fp2);
                if(count != blen)  return;
                memcpy(initial, ini_len, llen1_M);
                memcpy(ini_len1, code, llen_M);
                des(code, bufo);
        }
}
void cfb_e()
{
        ini_shift();
        des(code, bufo);
        while(count = fread(buf, SC, blen, fp1))
        {
                xor(bufo);
                fwrite(bufo, SC, blen, fp2);
                if(count != blen)  return;
                byte_to_code(bufo, cipher, blen);
                memcpy(x, xb1, blen8a_M);
                memcpy(xb2, cipher, blen8_M);
                memcpy(initial, ini_len, llen1_M);
                memcpy(ini_len1, x, llen_M);
                des(code, bufo);
        }
```

```
}
void cfb_d()
{
        ini_shift();
        des(code, bufo);
        while(count = fread(buf, SC, blen, fp1))
        {
                byte_to_code(buf, cipher, count);
                memcpy(x, xb1, blen8a_M);
                memcpy(xb2, cipher, blen8_M);
                xor(bufo);
                fwrite(bufo, SC, blen, fp2);
                if(count != blen)  return;
                memcpy(initial, ini_len, llen1_M);
                memcpy(ini_len1, x, llen_M);
                des(code, bufo);
        }
}
void usage(prog)
char *prog;
{
        fprintf(stderr, "Usage : %s [-Xn] [-Kkey] [-Finit] [-Jn] [-Ln] <-E|-D>  file1
file2¥n", prog);
        fprintf(stderr, "    -X : Modes of Operations¥n");
        fprintf(stderr, "            n : Mode Number (default = 1)¥n");
        fprintf(stderr, "                1 =  ＥＣＢ(Electric CodeBook)¥n");
        fprintf(stderr, "                2 =  ＣＢＣ(Cipher Block Chaining)¥n");
        fprintf(stderr, "                3 =  ＯＦＢ(Output FeedBack)¥n");
        fprintf(stderr, "                4 =  ＣＦＢ(Cipher FeedBack)¥n");
        fprintf(stderr, "    -K : key file definition¥n");
        fprintf(stderr, "            key : key file name¥n");
        fprintf(stderr, "    -F : Initial Value of CBC,OFB,CFB mode(default =
all'0')¥n");
        fprintf(stderr, "            init : initial value file name¥n");
        fprintf(stderr, "    -J : Block length direction¥n");
        fprintf(stderr, "            n : Bleck length(1 - 8 Bytes, default = 8)¥n");
```

```
        fprintf(stderr, "     -L : Key Feedback bit length direction¥n");
        fprintf(stderr, "          n : Feedback length(1 - 64, default = 64)¥n");
        fprintf(stderr, "     -E : Encryption mode¥n");
        fprintf(stderr, "     -D : Decryption mode¥n");
        fprintf(stderr, "    file1 : source file¥n");
        fprintf(stderr, "    file2 : destination file¥n¥n");
        fprintf(stderr, "    (note 1 : file is 8byte Hexa-decimal code = 16 characters.)¥n");
        exit(0);
}
void openerr(kind, file)
char *kind, *file;
{
        fprintf(stderr, "Error : %s file(%s) can't open.¥n", kind, file);
        exit(-1);
}
int main(argc, argv)
int argc;
char *argv[];
{
        char *prog = argv[0];
        long tm0, tm1;
        int edflag = -1, ki = 0, kk = 0, tflag = 0;
        int k, k1;
#ifdef  DEBUG
        static char desmode[4][4] = {"ECB", "CBC", "OFB", "CFB"};
        setvbuf(stderr, (char *)NULL, _IONBF, 0);
        setvbuf(stdout, (char *)NULL, _IONBF, 0);
#endif
        if(argc < 2 || *argv[1] != '-')         usage(prog);
        k = 1;
        do
        {
                switch(tolower(*(argv[k] + 1)))
                {
                case 'x':  mode = atoi(argv[k] + 2);
                                        if(mode < 1 || mode > 4)  usage(prog);
```

```
                                        break;
            case 'k': kk = k;
                                        break;
            case 'f':  ki = k;
                                        break;
            case 'j':  blen = atoi(argv[k] + 2);
                                        if(blen < 1 || blen > 8)     usage(prog);
                                        blen8 = blen * 8;
                                        blen8_M = blen8 * sizeof(UINT);
                                        blen8a = 64 - blen8;
                                        blen8a_M = blen8a * sizeof(UINT);
                                        break;
            case 'l':  llen = atoi(argv[k] + 2);
                                        if(llen < 1 || llen > 64)     usage(prog);
                                        llen_M = llen * sizeof(UINT);
                                        llen1 = 64 - llen;
                                        llen1_M = llen1 * sizeof(UINT);
                                        break;
            case 'd': if(edflag == 1)     usage(prog);
                                        edflag = 0;
                                        break;
            case 'e': if(edflag == 0)     usage(prog);
                                        edflag = 1;
                                        break;
            case 't':  tflag = 1;
                                        break;
            case 'm': fp1 = fopen("des.man", "r");
                                        if(fp1 == NULL)  usage(prog);
                                        while(fgets(buf, 256, fp1) != NULL)
                                                fprintf(stderr, "%s", buf);
                                        exit(0);
            default: usage(prog);
            }
} while(*argv[++k] == '-');
if((mode == 4 && blen8 > llen) || edflag == -1 || k >= argc) usage(prog);
k1 = k++;
```

```
if(k >= argc)        usage(prog);
init_des();
if(kk != 0)
{
        fp1 = fopen(argv[kk] + 2, "r");
        if(fp1 == NULL)  openerr("key", argv[kk] + 2);
        fgets(buf, 256, fp1);
        fclose(fp1);
        hex_to_code(buf, key);
}
if(mode >= 2)
{
        if(ki)
        {
                fp1 = fopen(argv[ki] + 2, "r");
                if(fp1 == NULL)  openerr("initial", argv[ki] + 2);
                fgets(buf, 256, fp1);
                fclose(fp1);
                hex_to_code(buf, initial);
        }
        if(mode == 2)       bufo_blen = bufo + blen;
        if(mode >= 3)
        {
                ini_len    = initial + llen;
                ini_len1 = initial + llen1;
        }
        if(mode == 4)
        {
                xb1 = x + blen8;
                xb2 = x + blen8a;
        }
}
fp1 = fopen(argv[k1], "r");
if(fp1 == NULL)  openerr("source", argv[k1]);
fp2 = fopen(argv[k], "r");
if(fp2 != NULL)
```

```
        {
                fprintf(stderr, "Error : destination file(%s) already exist.¥n", argv[k]);
                exit(-1);
        }
        fp2 = fopen(argv[k], "w");
#ifdef  DEBUG
        if(edflag)              printf("<<< Encryption (");
        else                    printf("<<< Decryprion (");
        printf("%s) >>>¥n", desmode[mode - 1]);
        printf("    file1 : %s¥n", argv[k1]);
        printf("    file2 : %s¥n", argv[k2]);
        printf("     key    : ");
        cprint(key);
        putchar('¥n');
        if(mode > 1)
        {
                printf("initial : ");
                cprint(initial);
                putchar('¥n');
        }
        if(mode == 3)       printf("Block length = %d [Bytes]¥n", blen);
#endif
        tm0 = clock();
        switch(mode)
        {
        case 1:   makekey(edflag);
                        ecb();
                        break;
        case 2:   makekey(edflag);
                        if(edflag)          cbc_e();
                        else                cbc_d();
                        break;
        case 3:   makekey(1);
                        ofb();
                        break;
        case 4:   makekey(1);
```

```
                        if(edflag)          cfb_e();
                        else                cfb_d();
                        break;
        }
        tm1 = clock();
        fclose(fp1);
        fclose(fp2);
        if(tflag)  fprintf(stderr, "lap time : %7.3f [s]¥n", (double)(tm1 - tm0) * 1.e-6);
        return 0;
}
```