公立はこだて未来大学 2014 年度 システム情報科学実習
グループ報告書

Future University Hakodate 2014 System Information Science Practice

Group Report

プロジェクト名

素因数分解

Project Name

Prime Factorization

グループ名

グループ A

Group Name

Group A

プロジェクト番号/Project No.

8-A

プロジェクトリーダ/Project Leader

1012022　今井啄人　Takuto Imai

グループリーダ/Group Leader

1012022　今井啄人　Takuto Imai

グループメンバ/Group Member

1011191　上戸真裕　Masahiro Ueto
1012022　今井啄人　Takuto Imai
1012090　木村純平　Junpei Kimura
1012093　小濱拓也　Takuya Kohama
1012104　吉田努　Tsutomu Yoshida
1012171　狩野大樹　Taiki Karino
1012174　佐藤康太郎　Kotaro Sato
1012175　清水目佳樹　Yoshiki Shimizume

指導教員

白勢政明　由良文孝

Advisor

Masaaki Shirase　Fumitaka Yura

提出日

2015 年 1 月 14 日

Date of Submission

January 14, 2015

概要

本プロジェクトの目的はある整数のできるだけ大きい素因数を見つけることである．現在公開鍵暗号において RSA 暗号がよく使われている．RSA 暗号とは素因数分解の困難性を利用した暗号方式である．RSA 暗号は素因数分解をすることによって安全性の評価をすることができる．その素因数分解の最も優れた解法の一つに楕円曲線法（ECM）がある．私たちは楕円曲線法による素因数分解プログラムの高速化とアルゴリズムの改良を目指す．
また, 楕円曲線法を用いて大きい素因数を見つけることを目的とする ECMNET[1] というサイトがある．現在記録されている素因数よりも大きい素因数を見つけることで ECMNET のランキングに誰でも名前を載せることができる．私たちはランクインしている素因数を参考にし, より大きな素因数を発見したい．
大きい素因数を見つけるという目的を達成するために, アルゴリズムをよく理解しプログラムの高速化に繋がるような理論を学ぶ理論班, そして学んだ内容をプログラミングし理論班が見つけた高速化に繋がる手法を実装するプログラミング班に分かれて取り組んだ．
理論班の活動はプログラムの高速化を図るために, 楕円曲線法の計算量を削減することを目的に活動した．計算量を削減するために様々な計算手法を学び最も適切な手法を探した．逆元計算が乗算に比べ約 12 倍もの時間的なコストがかかってしまうため, 逆元計算を如何に減らすかということに重点を置き活動した．論文を輪読し射影座標系, Jacobian 座標系といった計算量を削減する手法を学んだ．より計算量を削減する手法を求め独自の計算法を試行錯誤しながら計算手法の改良を試みた．
プログラミング班の活動はより高速なプログラムの作成を目的に活動した．まず楕円曲線法を行うプログラムを開発するために, 任意精度演算ライブラリである GMP を導入した．複数人によるプログラミングをスムーズに行うためにプログラム仕様書やコーディング規約を定めた．射影座標系を用いることにより計算量を削減し, 高速化を図ることができるため, 射影座標系の計算手法を実装した．また, より因数の発見確率をあげるため, OpenMP を用いた並列処理を行った．OpenMP とはマルチスレッド並列プログラミングのための API である．これにより多くの異なる楕円曲線を並列実行することができるようになった．我々が作成したプログラムは Xeon Phi 上で実行している．Xeon Phi とは Intel 社で開発された 60 個のコアをもつコプロセッサーである．
このような理論班とプログラミング班の活動により楕円曲線法を用いたプログラムを作成することができた．

キーワード　楕円曲線法, 素因数分解, ECMNET, RSA 暗号, 公開鍵暗号, 射影写像系, Jacobian 座標系, GMP, OpenMP, Xeon Phi

（※文責: 狩野大樹）

Abstract

A purpose of our project is to find prime factors of an integer as big as possible. Recently, RSA cryptosystem is widespread all over the world. RSA is based on the hardness of prime factorization and then security of RSA can be evaluated by prime factorization. Elliptic curve method is one of the best method for solving prime factorization. We aim at speeding up a program and improving an algorithm of prime factorization.
ECMNET is a website about prime factorization challenge. Anyone can try it.Therefore, we challenge ECMNET ranking to find big factors. Anyone who get the bigger factor than before one can record his or her name on this site. We would like to discover a bigger prime than an ECMNET record.
In order to achieve our purpose, we were divided into two groups. First group is Algorithm group. Their activity was to increase the speed of the ECM. Therefore, they learned an important theories. Second group is Programming group. Their activities was to create an ECM program. The program used theory algorithm group made.
Algorithm group was intended to reduce the complexity of elliptic curve method. They learned the various computational techniques to reduce the amount of calculation. An inverse element calculation takes about 12 times temporal cost compared to multiplication. They worked to reduce it. They learned the calculation methods, Projective coordinate system and the Jacobian coordinate system, by reading a paper. In addition they tried to improve calculation methods.
Programming group's activity was to speed up the program. They introduced the GMP that is an arbitrary precision arithmetic library. They made the program specification and coding conventions for them to program smoothly. They implemented the calculation method of Projective coordinate system, because it is possible to accelerate calculation the arithmetic speed. The OpenMP is an API for parallel programming. They adopted the OpenMP to accelerate a computing speed.Both methods increase the discovery probability of factor. Many different elliptic curve methods programs can be executed in parallel. This programs were run on Xeon Phi. The Xeon Phi is a coprocessor which has 60 cores and is developed by Intel Corporation.
Therefore, we were able to create a program using the elliptic curve method by our groups activity.

Keyword Elliptic curve method, Prime factorization, ECMNET, RSA cryptosystem, Public key cryptosystem, Projective coordinate system, Jacobian coordinate system, GMP, OpenMP, Xeon Phi

（※文責: 狩野大樹）

目次

第 1 章　プロジェクトの目的

本プロジェクトの目的は, 楕円曲線法を用いてある整数のできるだけ大きい素因数を見つけることである.

（※文責: 佐藤康太郎）

1.1　プロジェクトの背景

現在公開鍵暗号において RSA 暗号がよく使われている. RSA 暗号とは素因数分解の困難性を利用した暗号方式である.

RSA 暗号は素因数分解をすることによって安全性の評価ができる. その素因数分解の最も優れた解法の一つに楕円曲線法（ECM）がある.

本プロジェクトではこの楕円曲線法を用いて素因数分解をする.

（※文責: 狩野大樹）

1.2　ECMNET の説明

”ECMNET”とは楕円曲線法を用いて, ある整数の大きい素因数を見つけることを目的とするコンペティションである. そこでは楕円曲線法の歴史や優秀さ, 今現在ランクインしている素因数の桁数を知ることが出来る. ランキングには約 60〜70 桁ぐらいの素因数がランクインしている.

現在記録されている素因数よりも大きい素因数を見つけることで ECMNET のランキングに発見者の名前を載せることができる.

（※文責: 狩野大樹）

1.3　プロジェクトの最終目標

今まで基礎学習で学んだ知識を用いて ECMNET に挑戦し, ランキング入りを目指す. また, 実装したプログラムの処理の高速化を目指す.

（※文責: 清水目佳樹）

第 2 章　課題

2.1　課題設定

本プロジェクトの目標は大きな数を楕円曲線法を使い素因数分解することである.
目標を達成するため, 我々は次の課題を建てることにした.

- ECM に関する基礎知識をつける, 主に代数学の基礎的な素養を身につける
- ECMNET のルールを理解する
- 既存の楕円曲線法を用いた素因数分解プログラムの高速化

（※文責: 上戸真裕）

2.2　課題解決のためのプロセス

2.2.1　ECM の基礎学習

楕円曲線法のアルゴリズムを学習するにあたり, プロジェクトメンバー全員が楕円曲線法のアルゴリズムを理解するための必要な知識が不足していた.

そこで, 前期は担当教員である白勢先生に準備していただいたカリキュラムに沿い基礎学習を行うことにした.

カリキュラムの内容は, 以下の通りである.

- 群, 環, 体について
- $\mathbb{Z}/n\mathbb{Z}$ について
- $\mathbb{Z}/n\mathbb{Z}$ での＋と×
- $\mathbb{Z}/n\mathbb{Z}$ のマイナス元と引き算
- $\mathrm{E}(\mathbb{R})$ の演算*
- $\mathbb{Z}/n\mathbb{Z}$ の逆元と割り算
- $\mathbb{Z}/n\mathbb{Z}$ の元が逆元を持つ条件
- $\mathbb{Z}/n\mathbb{Z}$ は環
- $\mathbb{F}_p$=$\mathbb{Z}/p\mathbb{Z}$ は体
- 有限群とラグランジュの定理
- $\mathbb{F}_p$ の元の平方根
- $\mathrm{E}(\mathbb{R})$ の演算＋
- スカラー倍
- 加算公式
- $\mathrm{E}(\mathbb{F}_p)$ について
- $\mathrm{E}(\mathbb{F}_p)$ にラグランジュの定理を適用
- $\mathrm{E}(\mathbb{Z}/n\mathbb{Z})$ と $\mathrm{E}(\mathbb{F}_p)$ との関係
- 素因数分解の楕円曲線法 (ECM)

（※文責: 今井啄人）

2.2.2 ECMNET の調査

ECMNET に挑戦するにあたって, まずは ECMNET がどのようなサイトであるのかを調査することにした. その結果, ECMNET は大きな素因数を見つけることを目的としたサイトだということが分かった.

また, ECMNET は英語のサイトであるので, 今後の作業を明確にするためにもまずは全員で ECMNET 内の重要と思われる部分の和訳を行った.

その後, プロジェクトメンバー全員で読み合いを行って内容を理解した.

（※文責: 小濱拓也）

2.2.3 既存のプログラムの高速化

現在のプログラムでも素因数分解は可能だが, 桁数がある程度まで増えると処理に時間がかかってしまう. そこで我々はハードウェア面で ECM を高速化することにした. 具体的には, ”Xeon Phi”というメニーコア・コプロセッサー上で ECM を動作させるためのプログラムを書くことで高速化を図る.

（※文責: 小濱拓也）

第 3 章　活動内容

3.1　前期

3.1.1　基礎学習

本プロジェクトでは, 楕円曲線法での素因数分解を実装し, ECMNET に挑戦することが目標である. 前期には楕円曲線法のアルゴリズムを理解するための基礎学習を行った. 担当教員の白勢先生に作成していただいたカリキュラムに沿い, 計算問題や証明問題に取り組んだ. 以下のような内容について学んだ.

（※文責: 木村純平）

1. 群, 環, 体

（a）群

ある演算 $\circ$ が定義されている集合 G が以下の条件を満たすとき $(G, \circ)$ は群であると言う.

i. 結合法則を満たす
ii. 単位元が存在する
iii. 任意の $a \in G$ に対して逆元が存在する

（b）環

2 つの演算 $+$ と $\times$ が定義されている集合 R が次を満たすとき, R を環と言う.

i. R は $+$ に関して計算順序を変えても計算結果が変わらない群 (アーベル群) である
ii. $\times$ に関して結合法則を満たす
iii. R は $\times$ に関する単位元を含む
iv. 分配法則を満たす

（c）体

環 K が

$K \setminus \{0\}$ の全ての元に対して $\times$ に関する逆元が存在する

とき, K を体という.

（※文責: 木村純平）

2. ラグランジュの定理

$(G, \circ)$ を $\#G = l$ であり単位元が e の有限群とする. すると, どんな $a \in G$ に対しても,

$$\underbrace{a \circ a \circ \cdots \circ a}_{l-1 \text{ 回の } \circ} = e$$

が成り立つ.

（※文責: 木村純平）

3. $\mathbb{Z}/n\mathbb{Z}$ について

自然数 n に対して, 0 から $n-1$ までの整数の集合を $\mathbb{Z}/n\mathbb{Z}$ と書く.

素数 p に対して, 0 から $p-1$ までの整数の集合を F_p と書くこともある.

$$\mathbb{Z}/n\mathbb{Z} = \{0, 1, 2, ..., n-1\}$$

（a）$\mathbb{Z}/n\mathbb{Z}$ での演算

i. $\mathbb{Z}/n\mathbb{Z}$ の加算

$a, b \in \mathbb{Z}/n\mathbb{Z}$ に対して,

$$a + b \ (\mathbb{Z}/n\mathbb{Z} \text{ での加算}) := (a + b) \bmod n$$

とする.

ii. $\mathbb{Z}/n\mathbb{Z}$ の乗算

$a, b \in \mathbb{Z}/n\mathbb{Z}$ に対して,

$$a \times b \ (\mathbb{Z}/n\mathbb{Z} \text{ での乗算}) := (a \times b) \bmod n$$

とする.

iii. $\mathbb{Z}/n\mathbb{Z}$ の減算

$a, b, c \in \mathbb{Z}/n\mathbb{Z}$ に対して,

$(a + b) \bmod n = 0$ となるような b を a のマイナス元という. マイナス元を用いて

$$c - a \ (\mathbb{Z}/n\mathbb{Z} \text{ での減算}) := (c + b) \bmod n$$

とする.

iv. $\mathbb{Z}/n\mathbb{Z}$ の除算

$a, b, c \in \mathbb{Z}/n\mathbb{Z}$ に対して,

$(a \times b) \bmod n = 1$ となるような b を a の逆元という. 逆元を用いて

$$c \div a \ (\mathbb{Z}/n\mathbb{Z} \text{ での除算}) := (c \times b) \bmod n$$

とする.

（b）$\mathbb{Z}/n\mathbb{Z}$ での逆元について

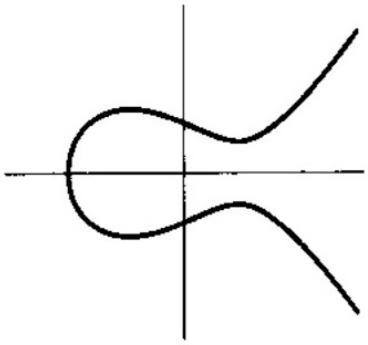
$a \in \mathbb{Z}/n\mathbb{Z}$ に対して $\gcd(a,n) \neq 1$ の時, a は逆元を持たない.

（※文責: 木村純平）

4. 楕円曲線

（a）$y^2 = x^3 + ax + b$ の形をした 3 次曲線 E を楕円曲線という.

（b）楕円曲線 E を実平面に書くと下図の様な x 軸に対称な曲線となる.



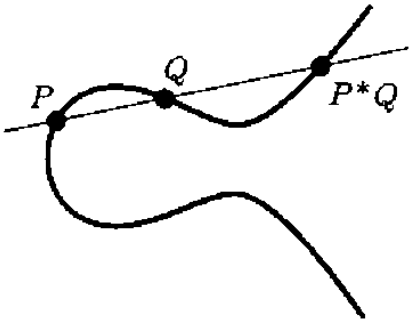
$\mathcal{O}$ を含む楕円曲線 E の点全体の集合を $E(\mathbb{R})$ と表す. $\mathbb{R}$ は実数全体の集合である.
なお, $\mathcal{O}$ は無限遠点とする.

（a）$E(\mathbb{R})$ の演算 $*$

i. 異なる 2 点 $P, Q \in E(\mathbb{R})$ をとる

ii. P と Q を通る直線を引く

iii. 第 3 の交点を $P * Q$ とする



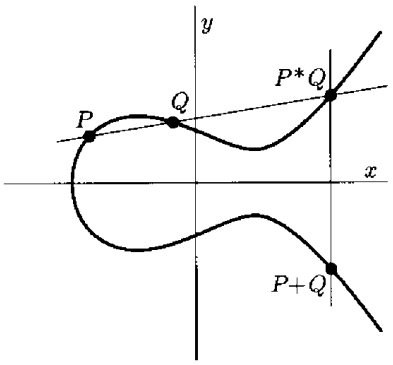
（b）$E(\mathbb{R})$ の演算 $+$

$P, Q \in E(\mathbb{R})$ に対して,

$$P + Q = (P * Q) * \mathcal{O}$$

と定義する.

$P * Q$ の x 軸に関して対称な点を $P + Q$ とする.



なお, $(E(\mathbb{R}), +)$ は $\mathcal{O}$ を単位元とする群となる.

（c）スカラー倍

$P \in E(\mathbb{R})$ と自然数 n に対して, nP を

$$nP = \underbrace{P + P + P + \cdots + P}_{n \text{ 個の和}}$$

と定義する.

（d）加算公式

楕円曲線 E 上の点 P, Q の座標が与えられている時 ($P = (x_1, y_1)$,$Q = (x_2, y_2)$ とする), $P + Q$ の座標 (x_3, y_3) を x_1, y_1, x_2, y_2 を使って計算できる.

i. $x_1 = x_2,\ y_1 = -y_2$ の場合 (Q は P の対称点)

この場合は $Q = -P$ なので,

$$P + Q = \mathcal{O}$$

となる.

ii. その他の場合

$P + Q$ は $\mathcal{O}$ とならないので, $P + Q$ は座標を持つ. $P + Q$ の座標を (x_3, y_3) とする.

A. λ の計算

$$\lambda = \begin{cases} \dfrac{y_1 - y_2}{x_1 - x_2} & x_1 \neq x_2 \text{の時} \\ \dfrac{3x_1^2 + a}{2y_1} & P = Q \text{ の時} \end{cases}$$

B. x_3 の計算

$$x_3 = \lambda^2 - x_1 - x_2$$

C. y_3 の計算

$$y_3 = \lambda(x_1 - x_3) - y_1$$

この式は加算公式と呼ばれる.

（※文責: 木村純平）

5. バイナリ法

バイナリ法とは一般的には累乗を高速で計算できる手法のことだが, スカラー倍にも適用できる. 係数の 2 進展開を用いてスカラー倍を計算する.

スカラー倍 nP の計算

n を 2 進展開したものを n[], 最下位桁を n[0], 最上位桁を n[m], 下位から数えて t 桁目を n[t] とし, 結果の値を格納する np を用意する.

初期値として np=P, t=m(最上位桁) を代入する.

(1)np=np+np
(2)n[t-1]=1 ならば np=np+P
(3)t=t-1

t>0 ならば (1) に戻る.
t=0 ならばスカラー倍 nP=np の計算を完了とする.

（※文責: 木村純平）

累乗の計算

累乗の場合は以下のアルゴリズムで計算を行う.

```
x = a;
for (i = t - 2; i  >= 0; i--) {
    x = x^2;
    if (b[i] == 1)
        x = x * a;
}
return x;
```

a, b の 2 進展開を受け取り, b のビットで, 最上位ビットを除くビットが 1 の時に元の a 倍を行っている. この計算によって, $t-1$ 回から $2t-2$ 回の乗算で a^b を計算することが出来る.

（※文責: 清水目佳樹）

3.1.2 ECMNET の和訳

ECMNET を利用するにあたり, ECMNET の和訳を行った. これにより我々は楕円曲線法の歴史, 楕円曲線法の優秀さ, ランキングに掲載されるための因数の桁数を学んだ. 私たちはランキングに名前を載せることができる素因数の桁数を把握し, 素因数分解を行う上で ECMNET を参考にすることが出来るようになった.

（※文責: 狩野大樹）

3.1.3 中間発表

ポスター制作

ポスター製作は中間発表の 1 ヶ月以上前から開始した. まず, 最初にポスターの構成を決定するために, 過去のポスター例を参考にした. 昨年度の最終発表で使用されたポスターをグループメンバー全員で鑑賞し, ポスターの構成を決定した. 発表は学生をターゲットとし, 本プロジェクトが何を目的としているか明瞭簡潔に伝えることを意識して文章を執筆した.

文章の執筆までは滞りなく完了したが, プロジェクトメンバー内にポスター制作を経験したことのあるメンバーがいなかったので, ポスターのデザインに苦戦した. ただ文章を並べるだけでは読みにくくなってしまうので, 見出しと本文のフォントの大きさや色のルールを決め, そのルールをポスター全体で守る様にした.

（※文責: 今井啄人）

スライド制作

中間発表準備序盤では, スライドは使わない予定だったが, ポスター発表だけでは非常に退屈である, と判断し楕円曲線法をスライドを用いて解説することを決定した. 数学用語は実例を使いながら解説し, 手順を明瞭にしながら解説することを意識して「Prezi」を用いてスライドを作成した. 「Prezi」とはクラウドサービスを使用した, プレゼンテーションソフトウェアである. 多様なアニメーション機能や視覚効果があり, オンラインでの共同編集ができるサービスである.

スライドのプロトタイプが完成し, プロジェクトメンバー全員で見たところ, このままでは理解し難いという意見がグループ内から出てきたので, 発表時にスライドと連携してホワイトボードを用いて実際に楕円曲線法で素因数分解を行うことにした. 発表時間の都合上, 比較的小さな整数「15」を使って素因数分解を行うことにした.

（※文責: 今井啄人）

発表練習

実演しながらスライドを使用するということで, スライドの内容も実演に合わせて変更する必要があった. 発表まで 2 週間しかなかったので, おおまかな台本を作成し, 発表練習をしながらスライドと台本の修正を行った.

前半発表班と後半発表班に分かれ, 相互評価を行い, 台本とスライドの改良を行った. 担当教員にも見てもらい, プレゼンの構成や改善点を指摘していただいた. 発表前日に完成させることができ

たが, 発表本番まで残された時間が少なく台本を暗記することが出来なかった.

（※文責: 今井啄人）

発表当日

当日の発表場所は本学 3 階モールで, 声が通りにくく, 声量が必要だった. 周りの他のプロジェクトも声を上げてプレゼンするため, 声が聞き取りにくくなってしまった.

ポスター制作, スライド, 台本の制作に準備期間の大半を費やしてしまったため, 質疑の準備を殆ど行うことが出来なかった. また, 予想される質問の回答を十分に準備していなかったので, スムーズに応対することが出来なかった.

（※文責: 今井啄人）

中間発表のアンケート結果

アンケートを集計した結果, 発表技術の平均点は 6.954 点, 発表内容の平均点は 7.027 点だった.

発表技術に関して目立ったコメントとしては, 原稿を見ていた, 話すスピードが速い, 例を用いてホワイトボードで計算していたのが良い, Prezi が良い, などがあった.

発表内容に関して目立ったコメントとしては, 目標が分かり辛い, 内容が難しい, 誰に対しての発表なのか分からなかったなどの意見があった.

（※文責: 清水目佳樹）

中間発表の反省

アンケートの結果から, 発表技術に関して, 台本を読んでいたり, 話すスピードが速いなどの問題点があったので, 後期の発表では聴講者を意識した発表を心掛けたい.

ホワイトボードや Prezi を用いた発表が好評だったので, 後期の最終発表会はより視聴者の興味を引く様な発表にする.

（※文責: 清水目佳樹）

3.2 後期

後期には理論班とプログラミング班に分かれて活動した.

（※文責: 上戸真裕）

3.2.1 理論班

目的

理論班の活動目的は楕円曲線法の計算量を削減することである. 計算量を削減することによってプログラムの高速化につながる. 具体的には加算公式, 2 倍算公式の計算のなかで行われる逆元計算の回数を減らすことを目的とした. これは逆元計算は乗算と比較すると約 12 倍もの時間的なコストがかかるためである.

（※文責: 狩野大樹）

活動内容

計算量削減のために様々な計算手法を学び一番適切なものはなんなのかを探した. 逆元の計算は乗算に比べて約 12 倍もの時間的なコストがかかる. よって如何に逆元計算の回数を減らすか, ということに重点を置いて活動した.

計算手法を学ぶために論文を輪読し, 射影座標系, Jacobian 座標系を学んだ. 学んだ内容から, より計算量を削減することができる手法を求め, 試行錯誤しながら加算公式の改良を試みた.

（※文責: 狩野大樹）

輪読

我々理論班は楕円曲線法（ECM）にシステム上用いる手法について, 数学的な理解を深めるため論文 [2] の輪読に勤めた. 論文を読み進める上で我々は射影座標系とヤコビアン座標系の 2 つの座標系に重点を置き, 勉強を進めることにした.

（※文責: 上戸真祐）

射影座標系

定義：アフィン座標系において次元が n であるとき, $(n+1)$ 個の「数」の比全体からなる空間

従来の手法で加法計算を行ったとき, 変数 (x, y) を定義し 2 次元のアフィン座標系で計算を行った. アフィン座標系から射影座標系への変換を行うため我々は次の変換を行った.

$$(x \quad y)^t \to (X \quad Y \quad Z)^t$$

ここであらわれた Z は任意に定まらなく 0 を含まない変数である. 次に同値関係を持つベクトルを定義する.

$$(X \quad Y \quad Z)^t \in F_p \times F_p \times F_p \backslash (0 \quad 0 \quad 0)$$

$$(X \quad Y \quad Z)^t \sim (\frac{X}{Z} \quad \frac{Y}{Z} \quad 1)^t$$

$(\frac{X}{Z} \quad \frac{Y}{Z} \quad 1)^t \to (x \quad y)^{\mathrm{t}}$ という対応関係があることがわかる, つまり $x = \frac{X}{Z}, y = \frac{Y}{Z}$ ということができるので従来の楕円曲線を次のように書き換えることができる.

$$Y^2Z = X^2 + aXZ^2 + bZ^3 (a, b \in K)$$

従来の手法に代入すると. PQ の 2 点をつなぐ直線の傾きは

$$\lambda = \frac{\frac{Y_1}{Z_1} - \frac{Y_2}{Z_2}}{\frac{X_1}{Z_1} - \frac{X_2}{Z_2}} = \frac{(Y_1Z_2 - Y_2Z_1)}{(X_1Z_2 - X_2Z_1)} = \frac{u}{v}$$

次に X_3,Y_3 を求める.

$$\begin{aligned}\frac{X_3}{Z_3} &= \frac{u^2}{v^2} - \frac{X_1}{Z_1} - \frac{X_2}{Z_2} \\ &= \frac{u^2Z_1Z_2 - v^2X_1Z_2 - v^2X_2Z_1}{v^2Z_1Z_2} \\ &= \frac{u^2Z_1Z_2 + v^3 - 2v^2X_1Z_2}{v^2Z_1Z_2}\end{aligned}$$

$$\begin{aligned}\frac{Y_3}{Z_3} &= \frac{u}{v}(\frac{X_1}{Z_1} - \frac{A}{v^2Z_1Z_2}) - (\frac{Y_1}{Z_1}) \\ &= \frac{u}{v}\frac{v^2X_1Z_2 - A}{v^2Z_1Z_2} - \frac{Y_1}{Z_1} \\ &= \frac{u(v^2X_1Z_2 - A) - v^3Y_1Z_2}{v^3Z_1Z_2}\end{aligned}$$

これより加算公式は次のように書き換えることができた.

$$\begin{aligned}X_3 &= vA \\ Y_3 &= u(v^2X_1Z_2 - A) - v^3Y_1Z_2 \\ Z_3 &= v^3Z_1Z_2\end{aligned}$$

ここで, $A = u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2$ である. 射影座標系は楕円曲線上の加法算において, 逆元計算の量を減らす方法として用いられる.

（※文責: 上戸真裕）

Jacobian 座標系

定義：射影座標系の拡張, 楕円曲線の方程式の次数を y の次数と x の次数の最小公倍数にそろえる.
有理数体において　$(x, y)^t \mapsto (\frac{X}{Y}^2, \frac{Y}{Z}^3)^t$ という対応をもつ, 有理点の座標を既約分数で書いたとき x の分母はある数の 2 乗であって, その数の 3 乗は y の分母に等しい.
よって, Jacobian 座標系で楕円曲線は次のようになる.

$$Y^3 = X^2 + aXZ^4 + bZ^6 \ (a, \ b \in R)$$

従来の手法に代入すると. PQ の 2 点をつなぐ直線の傾きは,

$$\lambda = \frac{(\frac{3X_1^2}{Z_1^4}) + a}{(\frac{2Y_1}{Z_1^3})} = \frac{3X_1^2 + aZ_1^4}{2Y_1Z_1}$$

$M = 3(X_1)^2 + a(Z_1)^4,\ S = 4X_1(Y_1)^2,\ T = M^2 - 2S$ と置く

$$\begin{aligned}
\frac{X_3}{Z_3^2} &= \frac{(3X_1^2 + aZ_1^4)^2}{4Y_1^2Z_1^2 - \frac{2X_1}{Z_2}} \\
&= \frac{(3X_1^2 + aZ_1^4)^2 - 8X_1Y_1^2}{4Y_1^2Z_1^2} \\
&= \frac{M^2 - 2S}{4Y_1^2Z_1^2} \\
&= \frac{T}{4Y_1^2Z_1^2} \\
&= \frac{T}{(2Y_1^2Z_1^2)^2}
\end{aligned}$$

$$\begin{aligned}
\frac{Y_3}{Z_3^3} &= \frac{3X_1^2 + aZ_1^4}{2Y_1Z_1}\left(\frac{-(3X_1^2 + aZ_1^4)^2 + 8X_1Y_1^2}{4Y_1^2Z_1^2} + \frac{X_1}{Z_1^2}\right) - \frac{Y_1}{Z_1^3} \\
&= \frac{3X_1^2 + aZ_1^4}{2Y_1Z_1}\left(\frac{-(3X_1^2 + aZ_1^4)^2 + 8X_1Y_1^2 + 4X_1Y_1^2}{4Y_1^2Z_1^2}\right) - \frac{Y_1}{Z_1^3} \\
&= \frac{(3X_1^2 + aZ_1^4)(-(3X_1^2 + aZ_1^4)^2 + 8X_1Y_1^2 + 4X_1Y_1^2) - 8Y_1^4}{8Y_1^3Z_1^3} \\
&= \frac{M(S-T) - 8Y_1^4}{8Y_1^3Z_13} \\
&= \frac{M(S-T) - 8Y_1^4}{(2Y_1^2Z_1^2)^3}
\end{aligned}$$

これより加算公式は次のように書き換えることができた.

$$\begin{aligned}
X_3 &= T \\
Y_3 &= M(S-T) - 8Y_1^4 \\
Z_3 &= 2Y_1Z_1
\end{aligned}$$

ここで $U_1 = X_1Z_2^2,\ U_2 = X_2Z_1^2,\ S_1 = Y_1Z_2^3,\ S_2 = Y_2Z_1^3,\ H = U_2 - U_1,\ r = S_2 - S_1$ である.

Jacobian 座標系は楕円曲線上の 2 倍算において, 逆元計算の量を減らす方法として用いられる.

（※文責: 上戸真裕）

加算公式の改良

射影座標系, Jacobian 座標系を通してさらなる計算量を削減する手法はないのかと試行錯誤し, 加算公式の改良を試みた. $x = \frac{X}{Z^2}$,$y = \frac{Y}{Z^2}$ や $x = \frac{X}{Z^3}$,$y = \frac{Y}{Z^3}$ などにして計算を行ってみた.

例として $x = \frac{X}{Z^2}$,$y = \frac{Y}{Z^2}$ で楕円曲線の加算を計算してみる.

$$\lambda = \frac{\frac{Y_1}{Z_1^2} - \frac{Y_2}{Z_2^2}}{\frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2}} = \frac{(Y_1 Z_2^2 - Y_2 Z_1^2)}{(X_1 Z_2^2 - X_2 Z_1^2)}$$

$$\begin{aligned} e &= Y_1 Z_2^2 - Y_2 Z_1^2 \\ f &= X_1 Z_2^2 - X_2 Z_1^2 \end{aligned}$$

$$\begin{aligned} \frac{X_3}{Z_3} &= \frac{e^2}{f^2} - \frac{X_1}{Z_1^2} - \frac{X_2}{Z_2^2} \\ &= \frac{e^2 Z_1^2 Z_2^2 - f^2 X_1 Z_2^2 f^2 X_2 Z_1^2}{f^2 Z_1^2 Z_2^2} \\ &= \frac{e^2 Z_1^2 Z_2^2 + f^3 - 2 f^2 X_1 Z_2^2}{f^2 Z_1^2 Z_2^2} \end{aligned}$$

$$B = e^2 Z_1^2 Z_2^2 + f^3 - 2 f^2 X_1 Z_2^2$$

$$\begin{aligned} \frac{Y_3}{Z_3} &= \frac{e}{f}\left(\frac{X_1}{Z_1^2} - \frac{C}{f^2 Z_1^2 Z_2^2}\right) - \frac{Y_1}{Z_1^2} \\ &= \frac{e}{f}\left(\frac{X_1 f^2 Z_2^2 - C}{f^2 Z_1^2 Z_2^2}\right) - \frac{Y_1}{Z_1^2} \\ &= \frac{e(X_1 f^2 Z_2^2 - C) - Y_1 f^3 Z_2^2}{f^3 Z_1^2 Z_2^2} \end{aligned}$$

$$\begin{aligned} X_3 &= fB \\ Y_3 &= e(f^2 X_1 Z_2^2 - B) - f^3 Y_1 Z_2^2 \\ Z_3 &= f^3 Z_1^2 Z_2^2 \end{aligned}$$

よって射影座標系よりもただ累乗計算が増えてしまっただけであった. 同様に $(\frac{X}{Z^3}, \frac{Y}{Z^3})$ で計算を行ってみても計算量削減には繋がらなかった.

（※文責: 狩野大樹）

3.2.2 プログラミング班

目的

プログラミング班の目的は, 前期に作成した楕円曲線法プログラムを元に, プログラムをより高速に実行できるよう改良することである.

（※文責: 今井啄人）

GMP 勉強会

GMP とは

GMP とは GNU Multi-Precision Library のことで, 任意精度演算のためのライブラリである. GMP を用いることで, メモリの上限を除いて, 事実上の桁数の制限無しに計算を行うことが出来る. GMP は主に, 暗号関連のアプリケーションやインターネットのセキュリティアプリケーション, コンピュータを用いた代数学の研究などを目的として開発された. GMP はフリーライブラリとして公開されている [4].

（※文責: 吉田努）

目的

楕円曲線法で因数分解を行う合成数の桁は数十桁, あるいは数百桁であるため, C 言語で用意されている型では演算を行うことが非常に困難である. そのため, 楕円曲線法を行うプログラムを開発するためには, 任意精度演算ライブラリである GMP を導入する必要があった.

（※文責: 吉田努）

勉強会

GMP の演算はすべて GMP が提供している関数を用いるため, 楕円曲線法を行うプログラム開発のためには GMP での演算になれる必要があった.

まず初めに, 全員で GMP のインストールを行った. GMP は演算を高速に行うために, 重要なループ部分などはアセンブリ言語で書かれている. そのため, 命令セットに命令が無いマシンもあり導入できない人もいたが, ほとんどの人が GMP をインストールすることが出来た. インストールする際, ライブラリのインストールを行うのは始めてと言う人が多かったため, インストール方法の確認も行った. インストールは Linux ユーティリティーの make を用いた. make はコンパイルを自動化するユーティリティーである. まずは, ライブラリに置いてある configure を実行した後, make install でプログラムのビルドを行った. その後, どのように演算を行うのかを知るため, GMP のホームページに掲載されている API をプログラミング班全員で読んだ. API を読んだ後に, 加算や乗算, 剰余算などの基本的な演算を用いた計算プログラムを書き, GMP での演算の練習を行った.

（※文責: 吉田努）

議事録

目的

プログラミング班内で話し合いが行われた際に, それらの内容を逐一記録することで,

後の製作活動の円滑に進めるために設けられた.

（※文責: 小濱拓也）

内容

実際に記載された内容は取り留めなく, 集合場所やコーディング規約に関するメモ, 試作プログラムの仕様や言伝など, 様々な内容が書き込まれた. ホワイトボードに板書が行われた際は, その内容を隈なく記録することで後の製作活動をサポートした. 些細な思いつきやコメントも無造作に書き込まれていたため, ある種の掲示板としての役割も得ていた.

（※文責: 小濱拓也）

プログラム仕様書

目的

普段用いない関数を多用するので, 複数人によるプログラミングをスムーズにするために作成した.

（※文責: 小濱拓也）

内容

内容は主に更新履歴, 概要, 関数一覧, マクロ一覧, 構造体一覧で成り立っていて, それぞれの機能や役割の説明を各項目ごとに記載した.

まず, 1 番最初に更新履歴を記載した. 更新した日付, 更新者, 更新した内容を記載した.

次に, 当プログラムの概要を記載した. 概要は極力専門用語を廃した上で記載された.

続いて, 関数の内容を記載した. 以下が記載された関数である. 「normal_add 関数」
:「P != Q」の場合に射影座標系にて加算を行う関数である.
「double_add 関数」
:「P == Q」の場合に射影座標系にて 2 倍算を行う関数である.
「ecm 関数」
:素因数分解を行う関数である.
「scalar 関数」
:スカラー倍の計算を行う関数である.
「afftopro 関数」
:直交座標系の点を射影座標系の点に変換する関数である.
「protoaff 関数」
:射影座標系の点を直交座標系の点に変換する関数である.

さらに, マクロを記載した. 以下が記載されたマクロである.
「affine_point_init」
:直交座標系の点 P を初期化するマクロ.
「affine_point_clear」
:直交座標系の点 P を開放するマクロ.
「affine_point_set」

：直交座標系の点 P を点 R に代入するマクロ.

「affine_point_cmp」

：直交座標系の点 P と点 Q を比較するマクロ.

：等しければ 0, 等しくなければ 1 を返す.

「projective_point_init」

：射影座標系の点 P を初期化するマクロ.

「projective_point_clear」

：射影座標系の点 P を開放するマクロ.

「projective_point_set」

：射影座標系の点 P を点 R に代入するマクロ.

「projective_point_cmp」

：射影座標系の点 P と点 Q を比較するマクロ.

：等しければ 0, 等しくなければ 1 を返す.

最後に, 構造体を記載した. 以下が記載された構造体である.

「projective_POINT」

：射影座標系の点を表す構造体.

「affine_POINT」

：直交座標の点を表す構造体.

そして, これらを以ってプログラム仕様書とした.

（※文責: 小濱拓也）

コーディング規約

目的

複数人でのプログラミングをスムーズにするために作成した.

（※文責: 小濱拓也）

内容

プログラムの命名規則やコーディングスタイル, その他重要と思われる点を網羅した. 演算子や制御文の記載方法等を規定した.

まず, 命名規則を定めた. 以下がその内容である.

- 変数の使用目的に沿った名前をつける
- 意味の区切りではアンダーバーを使用する
- 構造体名は大文字を使用する
- 関数は小文字を使用する
- ループ変数は小文字一文字を使用する

次に, コーディングスタイルを定めた. 以下がそのポイントである.

- 変数宣言
- 演算子
- 制御文
- 関数宣言

- インデント
- コメント

最後に, その他の重要項目を定めた. 以下がその内容である.

- 意味の区切りでは一行空白を入れる
- R = P + Q を function(R, P, Q) と表記する.
- プログラムを更新したら, どこを変更したのかをコメントに書く

そして, これらを以てプログラム仕様書とした.

（※文責: 小濱拓也）

コーディング

目的

プロジェクトの目的の一つである, 高速に動く素因数分解プログラムを作成するために行った.

（※文責: 小濱拓也）

内容

コーディング作業は複数人で行った. まず, 前期で学習した数式や, 新たに必要とされる数式を, GMP と呼ばれる多倍長計算を非常に高速に行うライブラリを参照してプログラミングすることになった. GMP を用いてプログラミングをする際は「x=x+1」という C 言語や Java で用いられている様な表記で計算・代入を行うことが出来ないため, 「mpz_add_ui(x, x, 1)」の様に 1 つ 1 つを専用の関数に直す必要があった. なお, この関数 1 つで計算と代入を同時に行っているため, 単純な計算の場合は必要とする行数は他の言語と大差は無い.

次に, 計算の途中で括弧を用いる式や, 1 以外の係数を持つ変数を用いて計算を行う場合に, 先にそちらを計算した結果を保存しておく temp 変数が必要となった. 例えば「x=x*(y+z)」という計算を行う場合, 先に「y+z」を行った結果を temp 変数に保存し, その temp 変数を x に掛け合わせ, その結果を x に代入するという手順を踏まなければならない. 式を変形して「x=x*y+x*z」とした場合でも, 「x*y」だけならば temp 変数無しで x に代入することが可能だが, 後半の「x*z」を保存する先が必要になるうえ, 先に行われた関数計算及び代入によって x の値が変わってしまうため, 計算結果が変わってしまうので, 計算トラブルを避けるためにも temp 変数を用いることとなった. 最終的に採用した数式は非常に長く続くものだったので, それに対応してプログラムも大量の変数や初期化, 解放が必要になった.

最後に, それらの宣言, 初期化, 関数計算, 解放を終えた上で, 各項目ごとにその項目に対応する説明文のコメントアウトを載せてコーディングを終了した.

（※文責: 小濱拓也）

テスト

目的

テストではコーディングした関数が正しく動作しているかの確認をテスト用プログラムを用いて行った. また, 前期に作成したプロトタイプの関数と比較して高速化が出来

ているか確認するためのテスト用プログラムも作成した．テストした関数は点の加算を行う normal_add(), double_add(), improved_normal_add(), jacobian_double() と，これらの関数を用いた scalar() である．

normal_add() は異なる射影座標系の二点を加算する関数で，double_add() は射影座標系の点を二倍算する関数である． improved_normal_add() は normal_add() と同じように異なる二点を加算するが射影座標系と直交座標系の点で加算を行う．jacobian_double() は Jacobian 座標系での点の二倍算を行う．scalar() は射影座標系の点のスカラー倍を計算する．

（※文責: 佐藤康太郎）

動作テスト

関数のテスト用プログラムは基本的に以下の流れになっている．

1. 初期点をランダムに生成
2. テストする関数で計算
3. プロトタイプ時の関数で計算
4. 結果を比較

1. から 4. を for 文によって多数ループさせて全ての結果が等しければ関数の動作が正確であるとした．

1. の初期点生成は get_affine_point() という関数を作成して行った．この関数はランダムに直交座標系の点を出力することが出来る．

2. はテストする関数によってプログラムを変更する必要があった．normal_add() をテストするときには get_affine_point() で生成される点は直交座標系であるため射影座標系に変換しなければいけなかった． 変換には afftopro() という関数を作り使用した．double_add() では normal_add() と同じようにして行うことが出来た．impoved_normal_add() は片方の点だけを射影座標系に変換してから計算を行った．jacobian_double() のときは afftojac() によって直交座標系から Jacobian 座標系へ点の変換を行ってから関数へ渡し，計算結果を jactoaff() によって直交座標系へ変換するという操作をした．scalar() のときはスカラー k をランダムに決定して，normal_add() のときと同じようにして射影座標系に変換してから関数に渡して計算をした．

3. は scalar() 以外の関数はほとんど変更する必要はなかった．normal_add(), double_add(), improved_normal_add(), jacobian_double() の場合は関数 add() で異なる二点の加算と二倍算の二通りを分けて計算した．add() は前期に作成したプロトタイププログラムで使っていた点の加算をする関数である．scalar() のときには add() を for 文で k 回ループさせて点の k 倍を計算させた．

4. は 2. と 3. で計算した結果の比較を行った．テストする関数で計算した結果は直交座標系に変換されているため，add() で計算した結果と座標ごとに比較して等しいかを調べた．

以上の流れで正しく動作しているかの確認を行った.

テストした結果は normal_add(), double_add(), jacobian_double(), scalar() は正確に計算できることは確認できたが, improved_normal_add() は正しく計算出来ていないことが確認できた.

（※文責: 佐藤康太郎）

速度比較

scalar() を直交座標系から射影座標系に変えたことで速度にどのぐらい変化があったかを見るためのプログラム scalar_time を作成した. この scalar_time は scalar() のテストに使用したテスト用プログラムを改良して作った. 射影座標系と直交座標系での scalar() の計算時間を time.h の関数 clock() を使用して計測した.

この結果, 射影座標系での scalar() の方が直交座標系での scalar() よりも最大２０%高速化していることが分かった.

（※文責: 佐藤康太郎）

射影座標系を用いた計算式の導入

前期の学習で私達は直交座標系での楕円曲線法を学んだ. しかし, 実際にプログラムに実装した場合, 除算が各所に絡んでしまうため, 計算コストが余計にかかってしまう. 除算の計算コストは乗算の計算コストに比べて 12 倍ものコストを消費するため, これは大きな問題になった. そこで私達は直交座標系だけではなく, 射影座標系を用いた計算式を導入することにした.

射影座標系とは, 従来の $(x \quad y)$ という 2 次元座標をそれぞれ $(\frac{X}{Z} \quad \frac{Y}{Z})$ とし, $(X \quad Y \quad Z)$ という 3 つの変数を用いて表現する手法のことである. こうすることで除算が行われる逆元計算の箇所を無くし, 計算コストを減らすことに成功した.

乗算のコストを M, 2 乗算のコストを S, 逆元計算のコストを I とした場合, 直交座標系と射影座標系それぞれの加算コストは以下の様になる. なお, S のコストは約 0.8M, I のコストは約 12M である.

- 直交座標系の加算：I+2M+S
- 射影座標系の加算：12M+2S

これらを全て M に直して比較すると, 直交座標系は約 14.8M, 射影座標系は 13.6M となり, 射影座標系の計算コストが直交座標系の計算コストを下回っていることがわかる.

理論上では約 1 割, 実際に導入しプログラムを動作させた結果では最大 2 割の高速化を実現することが出来た.

（※文責: 小濱拓也）

実装

目的

並列実行を行わず, 逐次実行するプログラムを製作していた段階では gcc でコンパイルしていた. しかし, mic アーキテクチャ (Intel 社のメニーコアプロセッサ Xeon Phi のアーキテクチャ) 向けにコンパイルするには, インテルが開発したコンパイラである icc を用いる必要がある. そのため, icc を用いたコンパイル方法を調べる必要があった. icc は基本的に gcc と同様のオプションを利用することができる. ライブラリの指定

は”-l”オプション, ライブラリの場所の指定には”-L”オプションを指定する. また, mic 向けにコンパイルするオプションである”-mmic”オプションと OpenMP を利用するための”-openmp”オプションを用いた. icc にこれらのオプションを付加することで, Xeon Phi 上で並列実行可能なプログラムをコンパイルすることが出来た.

```
\type{icc -mmic -openmp *.c -o funecm -lgmp -L/usr/local/lib/}
```

（※文責: 吉田努）

Makefile

プログラムに微妙な修正を施したい時や, 並列実行するスレッド数を変えたい時などが多々ある. その度に, コンパイルするためのコマンドを入力するのは大変不便である. そのため, Makefile を作りコンパイルを簡略化することにした. Makefile は make コマンドによって実行されるコンパイルを自動化するための手順書のようなものである. ターゲットとその依存関係を書き, コマンドを書く. そうすることで, 全体の依存関係から必要なターゲットに遷移し, コンパイルを行うことができる. Makefile によってコンパイル作業が簡略されたので, バグ修正の効率やプログラムの変更の効率が向上した.

（※文責: 吉田努）

実装されなかった手法

コーディングはしたが funecm プログラムに実装されなかった手法がいくつかあった. 加算を行う関数である improved_normal_add() や jacobian_double() である. imrpoved_normal_add() は射影座標系と直交座標系の点を加算して出力する関数である. しかし, テスト段階で点の座標が正確に計算されなかったことで, 現在実装されていない. ソースコード内での計算式に誤りがあると予想されるがいまだに発見できていない. もう一つ作成した関数である jacobian_double() は Jacobian 座標系での二倍算を行う関数である. Jacobian 座標系での二倍算は射影座標系や直交座標系での二倍算よりも高速に計算できるが, 二倍算でない普通の加算は他の二つの座標系よりも遅くなる. そのため, 二倍算だけを関数 jacobian_double() とし普通の加算を他の座標系としようとした. しかし, scalar() 内で二倍算から普通の加算に移行したときに座標系の変換をしなければならないため逆に計算コストが高くなってしまうため, 実装できなかった. 実装するには座標系の変換のところで工夫が必要である.

また, 高速フーリエ変換 (FFT) を使った因数分解法も実装することができなかった. 現在 funecm に実装されている楕円曲線法による因数分解を stage1, 高速フーリエ変換による誕生日攻撃に基づく因数分解を stage2 として実装できれば因数を発見する確率が向上することができる. stage2 は stage1 が因数を発見できなかった場合に実行される処理であり, stage1 で計算された結果を用いて実行される. そして, stage2 も楕円曲線上で計算される因数分解法なので, 現在のプログラムに組み込むことは難しくない. しかし, stage1 のコーディングに時間がかかってしまい stage2 を実装することが出来なかった.

（※文責: 佐藤康太郎）

性能評価

楕円曲線法は因数の大きさに依存する因数分解法であるため, 2 つの素数をかけ合わせた合成数を因数分解することで, 性能を評価することができる. 36 桁の素数とそれ以上の素数をかけ合わせた合成数を因数分解を行った結果, 36 桁の素数を発見することが出来た. この 36 桁の素数を発見するのにかかった時間は 1 時間程度であるため, より長時間プログラムを

稼働することで, より大きい素因数を見つけることが期待できる.

（※文責: 吉田努）

並列処理

プログラム班で作成したプログラムは最終的に Xeon Phi 上で並列実行し, 素因数を見つける. 並列実行するために OpenMP を導入した.

（※文責: 佐藤康太郎）

目的

楕円曲線法で因数分解をしても必ずしも因数が見つかるとは限らない. そのため, 多くの異なる楕円曲線で楕円曲線法を並列実行して因数を発見する確率を向上させることを目的としている.

（※文責: 佐藤康太郎）

Xeon Phi とは

一般的に Xeon Phi とは Intel 社で開発された 60 個のコアをもつコプロセッサーである. さらに, 最大 240 スレッドの処理が可能である. 今回, 使用していた Xeon Phi は 60 個のコアを持つコプロセッサー５１１０ P が二つ, ６４ GB のメモリ, 1 TB SATA HDD が搭載されている. また, OS は Red Hat Enterprise Linux v 6 がインストールされている. このように, Xeon Phi は並列コンピューティングやプログラミング開発マシンとしての高い能力が備わっている.

（※文責: 佐藤康太郎）

OpenMP

OpenMP とはマルチスレッド並列プログラミングのための API である. マルチコア CPU のコアを利用して複数のスレッドを処理している. 各スレッドは並列処理開始時に生成され処理が終了するとマスタースレッドのみの実行となる. 主に C 言語や Fortran 言語といったプログラミング言語で用いられる.

例えば実際に C 言語で for 文を並列化する場合には次のように書く.

```
#pragma omp parallel
{
#pragma omp for
for(i=0;i<100;i++)
{
  処理
}
}
```

`#pragma omp parallel` で囲んだ処理が並列化の対象となり, `#pragma omp for` で囲んだ for 文で i の異なる値で処理が並列実行される. また, 処理に使っている変数はスレッド全体で共有したり各スレッド内だけ有効になるようになどの設定も出来るようになっている.

変数の共有は`#pragma omp parallel shared(x, y)` とすると x, y は各ス

レッドで共有される． また, 変数を各スレッド内だけで有効にする場合には `#pragma omp parallel private(x,y)` とするとｘ, ｙのスコープがそのスレッド内だけでのみ有効となる．OpenMP については参考文献を参照した [3].

（※文責: 佐藤康太郎）

プログラムの並列化

楕円曲線法は楕円曲線が決定されれば独立してプログラムを実行することができる．そのため, 並列処理では異なる楕円曲線を構成して, それを各スレッドに渡し楕円曲線法による因数分解を並列実行している．

funecm では各スレッドに異なる楕円曲線を渡すために楕円曲線のパラメータである a をカウンタ変数として for 文を構成し, 並列実行を行っている．スレッド数は Xeon Phi 上での最大スレッド数である２４０としている．

```
#pragma omp parallel num_threads(omp_get_max_threads()) //最大スレッド数に設定
{
#pragma omp for
for(a=1;a<10000;a++){
楕円曲線法による因数分解
}
}
```

（※文責: 佐藤康太郎）

時間計測

各スレッドの実行時間とプログラム全体の実行時間を測るために時間計測の機能も funecm には付いている．OpenMP で並列化されているプログラムの時間を正確に測るためには OpenMP 専用の関数である omp_get_wtime() を使用する必要がある．最初は time.h の clock（）関数で時間計測をしようとしたが, 上手く計測されなかった．clock() 関数で計測できない理由は分からなかったが OpenMP について調べた結果, 専用の関数があることが分かった．omp_get_wtime() は double 型の値を返す関数である．計測開始位置と計測終了位置の二つで関数を呼び出し, その差を計算することで開始位置から終了位置までの時間を計測することが出来る．funecm では並列化を施している内部の最初から最後までの時間を計ることで各スレッドの時間を計測している．プログラム全体の実行時間も main 関数の最初から最後までが計測されている．
実際のプログラムとしては以下のような形式になる．

```
start=omp_get_wtime();
//処理
end=omp_get_wtime();
time=end-start;
```

start, end, time は double 型の変数である．start に処理開始時の時間を保持して, end に処理終了時の時間が保持されている．そして, t 開始時間と終了時間の差を計算し, 処

理時間を算出させて total に出力している．funecm ではこれを各スレッドが処理したときに表示させ, プログラムの最後に全体の処理時間が表示される．

（※文責: 佐藤康太郎）

スレッド数による速度比較

スレッド数によってプログラムの実行時間が変わるため, どのぐらいのスレッド数が良いのか調べるため実験を行ってみた．

スレッド数はプログラム内で設定することが出来き, 下記のようにすればできる．

```
#pragma omp parallel num_threads(スレッド数) {
  処理
}
```

最大スレッド数は `omp_get_max_threads()` を使うことによって取得できる．

実験は発見する因数の桁数を 20 桁として, それが見つかるまでの時間がスレッド数によってどのぐらい変わるのかを見た．まず, コア数と同じになるように 60 スレッドとして実行してみた結果, 約 10 秒となった．次に, 最大スレッド数である 240 スレッドとして実行してみると約 30 秒であった．しかし, 発見する因数の桁数を 36 桁にした場合には結果が異なった．60 スレッドの時には 1 時間以上実行し続けても発見できなかったが, 240 スレッドにすると 1 時間程度で発見することができた．

また, 入力する k の値によって因数が発見される回数がどのように変化するのかも見た．上記で述べた実験と同じように発見する因数の桁数を２０桁としたときで行った．k の値によって, いくつの楕円曲線で因数を発見できるのかを見た．結果としては k の値が約 1,000,000 だと 7 個の楕円曲線で因数を発見することが出来た．しかし，k の値が 1,000,000 未満だと 3， 4 個の楕円曲線でしか因数を発見することは出来なかった．これは，k の値を大きくすることで多くの楕円曲線で因数を発見できるようになるということを示している．

以上の結果から発見する因数が大きいときにはスレッド数が多い方が因数を発見できる確率が高くなり実行時間も短くなることが分かる．さらに，k の値を大きくすることでも因数の発見確率が向上されることも見れる．そのため, 現在 funecm でのスレッド数の設定は 240 となっている．しかし，k の値は大きくしすぎると実行時間が大幅に増加してしまうため決まった値にはしていない．

（※文責: 佐藤康太郎）

運用

Xeon Phi の利用

Xeon Phi を使用するには, ssh で仮想 OS にアクセスする必要がある．Xeon Phi は現在２つサーバに搭載されているため　各々, 以下のようにアクセスする．

```
$ ssh shirase@mic0
$ ssh shirase@mic1
```

mic は Xeon Phi 標準の名称で 0 から始まる．Xeon Phi を増設するごとに, 数字が上

がっていく．アクセスする際にはパスワードを求められるため, パスワードを入力し, アクセスを行う．

（※文責: 吉田努）

プログラムの実行準備

また, プログラムの実行に必要なライブラリやプログラム本体を仮想 OS 上にコピーして初めて Xeon Phi 上でプログラムを実行することが出来る．リモート先へのコピーには scp コマンドを用いた．前述したように, Xeon Phi は 2 つあり, コピー先は mic0 と mic1 の 2 つである．

```
$ scp /opt/intel/composerxe/lib/mic/libiomp5.so funecm
  .profile shirase@mic0
$ scp /opt/intel/composerxe/lib/mic/libiomp5.so funecm
  .profile shirase@mic1
```

上記のコマンドは, OpenMP のライブラリである”libiomp5.so”と因数分解プログラム本体である”funecm”, そして ssh 先で動的ライブラリを使用可能にする

```
$ export LD_LIBRARY_PATH=/home/shirase/
```

が書いてあるシェルの設定ファイルである”.profile”をコピーしている．

（※文責: 吉田努）

プログラムの実行

プログラムの実行は次の形式で行う．

```
$ ./funecm [composite] [k] > [file] &
```

第 1 引数には因数分解したい合成数を入力し, 第 2 引数には楕円曲線法のパラメータである k を入力する．$>$ はリダイレクトを表し, 標準出力をファイルに出力する．& はバックグラウンド実行を表す．

（※文責: 吉田努）

結果の確認

funecm は楕円曲線法を 240 並列で行っており, 長時間プログラムを稼働しているため, ログファイルが膨大な行数になってしまう ので, 目視で結果を探すのは困難である．そのため, linux ユーティリティプログラムである”grep”を用いて結果を確認している．

```
$ grep “factor found” [file]
```

このコマンドによって”factor found”という行を file から抜き出す事が出来る．つまり, もし, 因数分解出来ていたら grep によって結果が出力され, 因数分解出来ていなかった場合は何も表示されない．

（※文責: 吉田努）

ECM プログラムのマニュアル

目的

本プロジェクトで作成した ECM プログラムは, 利用方法が分かりにくく, プログラミング班の中でも, その利用方法を理解しているのは数人しかいなかった. マニュアルは次のプロジェクトでも利用することを想定して作成した. マニュアルはプログラムの使い方を詳解したドキュメントと, ハードウェアの使い方と全般的なプログラムの使い方を紹介している動画の 2 種類に分かれている. パソコンに不慣れな学生でも理解できるよう, 手順を一つ一つ解説している. 最後に, ECMNET の見方について解説した.

（※文責: 今井啄人）

ドキュメント

初めに, ハードウェア面での Xeon Phi を搭載したマシンの使い方を掲載した. マシンの起動方法や, UPS(無停電装置) との接続方法などを解説した. また, Xeon Phi と OS の関係などを解説している. LInux 環境に不慣れな人でも分かるように, プログラムを実行するために必要なコマンドを網羅した.

（※文責: 今井啄人）

動画

動画では, マシンのセッティングの仕方を解説した前編と, プログラムの使い方を説明した後編を作成した.

（※文責: 今井啄人）

ECMNET の見方

ECMNET は英語でかかれているため, 一目でどのような記録が掲載されているのかを把握するのは難しい. また ECMNET 独特な表現や, 専門用語などが登場するため分かり辛い. そのため, ECMNET の見方についての解説を行った.

（※文責: 今井啄人）

まとめ

ECM プログラムを高速に実行するために, 大きく 2 つのことを行った. まず一つ目に, 射影座標系での計算をモジュール化した. モジュールが完成するまでには仕様やコーディング規約を決定をした. さらに, コーディングが終わってからも動作が正確かを確認するために動作テストも行った. 二つ目に, Xeon Phi 上での並列処理を可能にした. 並列処理を可能にするために OpenMP を使用している. これにより 240 スレッドで並列処理を行えるようになった.

以上の二つからプログラムの高速化することができた. 性能評価を行った結果から, 前期に作成したプロトタイププログラムよりも速く大きな桁数の素因数を発見できることが分かった. 時間の都合上, 実装までにいたらなかった手法があったが, プログラミング班の目的は概ね達成できたと言えるだろう.

さらに, プログラムが完成してからはマニュアルを作成し, 誰でもプログラムをしようできるようにした. マニュアルには動画とドキュメントの二つの種類を用意して, 分かりやすいようにした. 次年度のプロジェクトで, このマニュアルを参考にすると良いだろう.

（※文責: 今井啄人）

3.2.3 最終発表

発表準備

後期は前期とは違い理論班とプログラミング班とに分かれて活動していたので発表もそれに合わせて構成した．ポスターはメインとサブポスター 2 枚の合計 3 枚を作成した．メインポスターは前期のものをベースに後期の活動内容を追加した．サブポスターは班ごとに作成した．理論班のサブポスターは, 理論班の活動のメインだった射影座標系での楕円曲線加算についてまとめ, プログラミング班のサブポスターは, プログラミング班の活動をまとめたものとした．

発表の形式は前期の中間発表とは異なりスライドのみにした．前期の中間発表のアンケートに「視線の移動が多い」という意見があったためである．発表のスライドは出来るだけシンプルで分かりやすいものになるようにした．その際, 出来るだけ専門用語を使わずに概要を説明できるようにスライドを構成したが, どうしても解説しなければならないような用語のみ解説することとした．そのような専門用語も出来るだけ具体例を交え解説するようにはしたが, 未来大学のカリキュラム内での内容であれば解説はしない, という風に取り決めた．

（※文責: 木村純平）

発表当日

前期には質問に対してスムーズな受け答えができなかったが, 最終発表会では前期の反省を活かして, 質疑応答がスムーズにできた．聴講者が聞き易い声の大きさで発表できた．

（※文責: 木村純平）

最終発表のアンケート結果

アンケートを集計した結果, 発表技術の平均点は 6.653 点, 発表内容の平均点は 7.041 点だった．前期と比べると発表技術の平均点は 0.3 点ほど低く, 発表内容の平均点はほぼ同じ点だった．

発表技術に関して目立ったコメントとしては, 原稿を見ないようにしたほうが良い, 用語などが分かりづらいので説明が必要だと思う, スライドが良い, などがあった．

発表内容に関して目立ったコメントとしては, 難しい内容ではあったが目標が理解できた, 31 桁がどのくらいすごいのか分からなかった, などがあった．

（※文責: 木村純平）

最終発表の反省

前期の反省にもあるが, また原稿を過度に見ながらの発表になってしまった．もう少し発表練習の時間を多くとっていれば, 原稿を暗記してより良い発表ができたと考えられる．

事前に用語に関する説明をあまりしないよう取り決めでいたが, アンケートでは用語の解説が必要だ, という意見が多かった．なので, 用語の説明を足すべきだった．

（※文責: 木村純平）

第 4 章　成果

プロジェクトでは, 基礎学習の成果として ECM を用いたプログラムを作成した.

（※文責: 上戸真裕）

4.1　前期

4.1.1　基礎学習の成果

群について

演算 $\circ$ について閉じている集合 G に対して次の 3 条件を満たしている時, $(G, \circ)$ は群であるという.

1. 結合法則
   どんな $a, b, c \in G$ に対しても $(a \circ b) \circ c = a \circ (b \circ c)$
2. 単位元の存在
   どんな $a \in G$ に対しても $a \circ e = e \circ a$ となる $e \in G$ が存在する.
3. 逆元の存在
   どんな $a \in G$ に対しても $a \circ a' = a' \circ a = e$ となる $a \in G$ が存在する.

ここで, 群 $(G, \circ)$ が任意の $a, b \in G$ に対して

$$a \circ b = b \circ a$$

を満たすとき $(G, \circ)$ をアーベル群, または可換群という.

（※文責: 清水目佳樹）

環について

2 つの演算「$+, \times$」について閉じている集合 R が

1. R は $+$ に関してアーベル群である.
2. 任意の $a, b, c \in R$ に対して, $(a \times b) \times c = a \times (b \times c)$ を満たす.
3. R は $\times$ に関する単位元を含む.
4. 任意の $a, b, c \in R$ に対して

$$a \times (b + c) = a \times b + a \times c$$
$$(a + b) \times c = a \times b + b \times c$$

の 4 つの条件を満たすとき, R は環であるという.

（※文責: 清水目佳樹）

体について

環 K が

$$K\backslash\{0\} \text{ のすべての元に対して } \times \text{ に関する逆元が存在する}$$

とき, K を体という.

（※文責: 清水目佳樹）

$\mathbb{Z}/n\mathbb{Z}$ について

a を b で割った余り（剰余）を

$$a \ mod \ b$$

と表す.

自然数 n に対して, 0 から $n-1$ までの整数の集合を $\mathbb{Z}/n\mathbb{Z}$ と表す.

$$\mathbb{Z}/n\mathbb{Z} = \{0, 1, 2, \cdots, n-1\}$$

$\mathbb{Z}/n\mathbb{Z}$ は環の 4 つの条件を満たすため, 環であるということがわかる. $\mathbb{Z}/n\mathbb{Z}$ での加算は普通に加算を行ってから n での剰余をとる.

$$(a+b) mod \ n$$

$\mathbb{Z}/n\mathbb{Z}$ での乗算も加算と同様, 普通に乗算してから n での剰余をとる.

$$(a \times b) mod \ n$$

$\mathbb{Z}/n\mathbb{Z}$ での減算はマイナス元を用いる. マイナス元とは a に対して, $a+b=0$ となる b のことである. 減算 $a-b$ を a+(b のマイナス元) とし計算してから n での剰余をとる. ここでは b のマイナス元を c とする.

$$(a-b) mod \ n = (a+c) mod \ n$$

$\mathbb{Z}/n\mathbb{Z}$ での除算は逆元を用いる. 逆元とは a に対して, $a \times b = 1$ となる b のことを指す. しかし $\mathbb{Z}/n\mathbb{Z}$ では必ずしも逆元が存在するとは限らない. $\mathbb{Z}/n\mathbb{Z}$ の元が逆元を持たない条件は, $a \in \mathbb{Z}/n\mathbb{Z}$ である a が n と互いに素でないときである. つまり, a と n での約数が存在する場合には a の逆元は存在しないということである. 除算 $a \div b$ を a×(b の逆元) とし計算してから n での剰余をとる. ここでは b の逆元を d とする.

$$(a \div b) mod \ n = (a \times d) mod \ n$$

$\mathbb{Z}/p\mathbb{Z}$ の p が素数の場合, 0 以外のすべての元に逆元を持つため, 体の条件を満たす. 体であることを強調するため, $\mathbb{Z}/p\mathbb{Z}$ を $\mathbb{F}_p$ or $GF(p)$ と表すことがある. $\mathbb{F}_p$ は有限集合の体なので, 有限体という.

（※文責: 清水目佳樹）

$E(\mathbb{R})$ の演算 $+$

1. 楕円曲線の点の演算「+」
   P, Q を $E(\mathbb{R})$ の元として,

$$P + Q = (P * Q) * \mathcal{O}$$

   と定義する. $P + Q$ を求める操作は次のようになる.
   (a) L を P と Q を通る直線（$P = Q$ のときは P での接線）とする. E と L の第 3 の交点を $P * Q$ とする.
   (b) $P * Q$ と $\mathcal{O}$ を通る直線を V とする. （V は $P * Q$ を通る x 軸に平行な直線である. ）
   E と V のもう 1 つの交点を $P + Q$ とする.

（※文責: 清水目佳樹）

2. $P + \mathcal{O}$
   上記では楕円曲線上の点である P, Q の $P + Q$ を述べたが, ここでは片方が $\mathcal{O}$ であるときを考える. 例えば $Q = \mathcal{O}$ のとき, $P + Q$ がどうなるかを考える. 定義通り

$$P + \mathcal{O} = (\mathcal{P} * \mathcal{O}) * \mathcal{O}$$

   です. まず, $P * \mathcal{O}$ を考える. L を P と $\mathcal{O}$ を通る直線, つまり P を通る y 軸に平行な直線とする. $P * \mathcal{O}$ は E と L ともう一つの交点だが, これは P の x 軸に関する対称点となる. $P = (x, y)$ とすると

$$P * \mathcal{O} = (x, -y)$$

   となる. $P * \mathcal{O}$ が分かったため, 次に $P * \mathcal{O} =$（$\mathcal{P} * \mathcal{O}) * \mathcal{O}$ を考える. これまでと同様に, ($P * \mathcal{O}) * \mathcal{O}$ を求めるには, $P * \mathcal{O}$ と $\mathcal{O}$ を通る直線を考える. これは先ほど考えた直線 L と一致するが, あくまで $P * \mathcal{O}$ と $\mathcal{O}$ を通る直線と考える. $P + \mathcal{O} = (\mathcal{P} * \mathcal{O}) * \mathcal{O}$ は E とこの直線のもう一つの交点ですが, これは P である.

$$P + \mathcal{O} = \mathcal{P}$$

   これを見ると $\mathcal{O}$ と数字の 0 は似ている性質をもっていることが分かる.

（※文責: 清水目佳樹）

3. マイナス元
   楕円曲線上の点である P, Q に対して, $P = (x, y), Q = (x, -y)$ のときの $P * Q$ を考える. つまり, Q は P の x 軸に関する対称点. $P + Q = (P * Q) * \mathcal{O}$ であるため, まず $P * Q$ を考える. P と Q を通る直線 L は P を通る y 軸に平行な直線となる. $P * Q$ は E と L の第 3 の交点だが, これは $\mathcal{O}$ である.

$$P * Q = \mathcal{O}$$

   次に, $P * Q(= \mathcal{O})$ との $\mathcal{O}$ を通る直線 V を考えるが, これは $\mathcal{O}$ での接線である. よって, $P + Q$ は $\mathcal{O}$ での接線と E との「もう 1 つの交点」である. 実は V は $\mathcal{O}$ での 3 重接線である. これは射影平面によって分かる. 以上より, 「もう 1 つの交点」$= \mathcal{O}$ となる. つまり

$$P + Q = \mathcal{O}$$

$\mathcal{O}$ は 0 に似ていることから

$$Q = -P$$

と書く.

（※文責: 清水目佳樹）

加算公式

楕円曲線 E 上の点 P, Q の座標が与えられている時 ($P = (x_1, y_1)$,$Q = (x_2, y_2)$ とする), $P + Q$ の座標 (x_3, y_3) を x_1, y_1, x_2, y_2 を使って計算する.

1. $x_1 = x_2,\ y_1 = -y_2$ の場合 (Q は P の対称点)
   この場合は $Q = -P$ なので,
   $$P + Q = \mathcal{O}$$
   となる.

2. その他の場合
   $P + Q$ は $\mathcal{O}$ とならないので, $P + Q$ は座標を持つ. $P + Q$ の座標を (x_3, y_3) とする.

（※文責: 清水目佳樹）

4.1.2 $E(\mathbb{Z}/n\mathbb{Z})$ と $E(\mathbb{F}_p)$ の対応関係

n を (素因数分解したい) 合成数, p を n の素因数とする.
$a, b \in \mathbb{Z}/n\mathbb{Z}$ を選んで, 楕円曲線

$$E : y^2 = x^3 + ax + b$$

を構成し, $(x, y) \in E(\mathbb{Z}/n\mathbb{Z})$ を 1 つとる.
各 $e \in \mathbb{Z}/n\mathbb{Z}$ に対して,

$$e_p = e \bmod p$$

と今回のみ書くことにする.

$$E_p : y^2 = x^3 + a_p x + b_p$$

は, $\mathbb{F}_p$ 係数の楕円曲線となる.
$(x, y) \in E(\mathbb{Z}/n\mathbb{Z})$ ならば, $(x_p, y_p) \in E_p(\mathbb{F}_p)$ となる.
$\because$) $(x, y) \in E(\mathbb{Z}/n\mathbb{Z})$ ならば,

$$y^2 = x^3 + ax + b \text{ in } \mathbb{Z}/n\mathbb{Z}$$

を満たす. 両辺の mod$p$ を取ると,

$$y_p^2 = x_p^3 + a_p x_p + b_p \text{ in } \mathbb{F}_p$$

となるので, (x_p, y_p) は $E_p(\mathbb{F}_p)$ の元である.
つまり, $P \in E(\mathbb{Z}/n\mathbb{Z})$ に対して, $P_p \in E_p(\mathbb{F}_p)$ が対応する.

（※文責: 上戸真裕）

4.1.3 ECM

n を素因数分解したい合成数とする.

1. $a, b \in \mathbb{Z}/n\mathbb{Z}$ を選んで, 楕円曲線
$$E : y^2 = x^3 + ax + b$$
を構成し, $P = (x, y) \in E(\mathbb{Z}/n\mathbb{Z})$ を 1 つとる.
2. (数学用語での) 適当な自然数 k を選ぶ.
3. 加算公式とバイナリ法を使って,
$$(k!)P$$
を計算する.
4. この計算の途中で, $\lambda = (y_1 - y_2)/(x_1 - x_2)$ or $\lambda = (3x_1^2 + a)/2y_1$ を求める部分が生じるが, $x_1 - x_2$ or $2y_1$ が正則でなければ (逆元を持たなければ) 計算は続行不可能となる.
5. しかし, 計算が続行不可能 $\Leftrightarrow \gcd(x_1 - x_2, n) \neq 1$ or $\gcd(2y_1, n) \neq 1$ なので, n の約数が求まる.
6. $(k!)P$ が計算できてしまったら n の約数は求まらないので, 1. からやり直す.

（※文責: 上戸真裕）

いつ続行不可能となるか？

p を n の素因数とする.

ステップ 1 で選んだ $P = (x, y)$ に対して, $P_p = (x_p, y_p) \in E_p(\mathbb{F}_p)$ とする.

$l = \#E_p(\mathbb{F}_p)$ の素因数分解を
$$l = p_1^{e_1} \cdot p_2^{e_2} \cdot \cdots \cdot p_t^{e_t} \tag{4.1}$$
とする.

もし,
$$p_i^{e_i} \leq k \quad (i = 1, 2, \cdots, t) \tag{4.2}$$
ならば, ラグランジュの定理より
$$(k!)P_p = \mathcal{O}$$
となる.

つまり, $(k!)P_p$ の計算過程の途中のどこかで $\mathcal{O}$ が生じる.

初めて $\mathcal{O}$ が生じるステップでの加算公式は

$x_{1p} = x_{2p}$ または $y_{1p} = 0$ となる.

このとき $x_1 - x_2$ または y_{1p} は p の倍数 ($\mathbb{Z}/n\mathbb{Z}$ の非正則元) となり, 続行不可能となる. ($\Rightarrow n$ の約数が求まる.)

（※文責: 上戸真裕）

プログラムの手順

ここではプロトタイプとして開発した楕円曲線法を用いた素因数分解のプログラムの解説をする.

Prime Factorization

まずは素因数分解したい合成数 n を決定する.

また, ここでは k の値は 10000 とした.

さらに, $k!$ をそのまま計算するのではなく, k を 10000 回ループさせている.

```
for (k = 2; k < 10000; k++)
```

このループ内でスカラー倍の計算を行う. つまり, $(k)P$ の計算を行うことになる.

そこで関数 scalar() を呼び出している.

scalar() では主にバイナリ法による計算を行っている.

まず k のビット長を変数 m に代入する. 値を 2 進数に変換し, それを配列に格納する.

```
  for (i = 0; i < m; i++)
{
  bit[i] = (k >> i) & 1;
}
```

バイナリ法は下記の方法で実行している.

```
mpz_set(tmp->x, P->x);
mpz_set(tmp->y, P->y);
i = m - 2;
  while (i > 0)
{
  add (P, P, P);
  if (bit[i] == 1)
    {
  add (P, P, tmp);
    }
  i--;
}
```

初めに, 変数 i にビット長から 2 引いた値を代入している. これは, 配列が 0 から始まり, 桁数は 1 から数えるためである.

また, k を 2 進数に変換したときのビットをチェックしていき, ビットが立っていれば元の点 P の加算を行う.

点の加算は関数 add () で行う.

今回, 任意精度長演算を行うために GNU Multi-Precision Library(GMP) を導入した.

また add() では, gcd () によって素因数を発見したかをチェックしている.

```
mpz_gcd (gcd, de, n);
if (mpz_cmp_ui (gcd, 1) != 0)
```

関数 `mpz_gcd()` と mpz_cmp_ui() は GMP の関数である.

mpz_gcd() は最大公約数を変数 gcd に代入し, mpz_cmp_ui() は gcd と 1 が等しいかをチェックしている.

（※文責: 吉田努）

計算結果

$10^{306}+1$ を合成数としてこのプログラムを動かした結果, 発見された最も巨大な素因数は 1575389803198166071 21(21 桁) であった.

（※文責: 吉田努）

考察

このプログラムでは高速化を施していないため, ECMNET のランキングに掲載されている様な巨大な素因数 (60 桁以上) を見つけることは, 現状では難しいと思われる.

そのため, アルゴリズムのと Xeon Phi への実装が必要である.

（※文責: 吉田努）

4.2 後期

4.2.1 理論班成果

掛け算の計算コストを M とし, 直交座標系と射影座標系の計算コストを比べると, 直交座標系での計算コストは掛け算の計算量の約 14.8 倍, 射影座標系での計算コストは掛け算の計算量の約 13.6 倍になった. よって, 計算コストを理論上 1 割削減できた.

（※文責: 清水目佳樹）

4.2.2 プログラミング班成果

ECM プログラムの改良点

射影座標系

前期のプロトタイププログラムでは加算を行うときに直交座標系の点を用いていたが今期に改良したプログラムでは射影座標系を採用した. 直交座標系の点を加算する場合には逆元計算を多用することになる計算コストが高い. しかし, 射影座標系の点で加算を行うと逆元計算をする回数が直交座標系のときよりも大幅に削減され計算コストを減らすことが出来る. 理論班が検証した結果, 理論的には直交座標系での加算と比較して射影座標系での加算は 10 %高速化されることが示された. 実際にプログラムを実行した結果では点のスカラー倍の計算時間を最大 20 %高速化することが出来ていた.

（※文責: 佐藤康太郎）

並列化

私達は Open-MP を用いて, プログラムの中で繰り返し計算する必要がある箇所をマルチスレッド化することに成功した. マルチスレッド化を施した最新の素因数分解プログラム (funecm 現行版) は, 既存の素因数分解プログラムよりも素因数を発見する確率が向上した.　マルチスレッド化を行い, スレッド数を Xeon Phi 上での最大スレッド数である 240 にした素因数分解プログラムはスレッド数が 60 の時の同プログラムより

も高速に素因数を発見することが出来た. 具体的には 36 桁の因数を発見する際, スレッド数が 60 の場合は 1 時間かけても素因数を発見するに至らなかったが, スレッド数が 240 の場合は約 1 時間で発見することが出来た. 桁数が少なくなると結果が逆転することも確認出来た. 実際に 20 桁の素因数を発見しようと, 前述のスレッド数 60 と 240 の同プログラムを実行した場合, 前者は約 10 秒で素因数を発見することが出来たが, 後者は発見までに約 30 秒かかってしまった. しかし今回目標としているのは 36 桁を遥かに超える 65 桁の素因数なので, スレッド数が多い方が効果的であることが判明した. よって, 最新の素因数分解プログラムではスレッド数 240 式を採用した.

（※文責: 小濱拓也）

改良した k!の計算

楕円曲線の点の個数が次のように素因数分解され,

$$\#E(F_p) = l = p_1^{e_1} \cdots p_t^{e_t}$$

と表す事が出来たとする.

$$p_i^{e_i} \leq k$$
$$(i = 1, 2, \cdots t)$$

上記の条件を満たす場合に, 楕円曲線上の点 P のスカラー倍が

$$(k!)P = O$$

となり, 計算続行不可能になるため, 因数を発見する事が出来るというものであった.
しかし, k!では合成数の累乗計算まで行っているため無駄が多い. そのため, 素数の累乗のみを計算するアルゴリズムを funecm では利用した.
以下に簡略化したアルゴリズムを掲載する

```
while (p < k) {
e = log k / log p;
for (i = 1; i <= e; i++) { /* e 回掛けることで e 乗を行う */
   scalar();   /* 点のスカラー倍を行う関数 */
}
p = nextprime(p);
}
```

最初は, $p^e <= k$ となるような e を取得している. 次に, p を e 回かけることで p^e を得ている.

（※文責: 吉田努）

ECM プログラムのマニュアル

概要

本プロジェクトで作成した ECM プログラムは, 使い方が分かりにくく, プログラミング班の中でもその使い方を知っているのは数人しかいなかった. そこで, 次年度のプロジェクトでも今年度作成したプログラムを利用し, 改良できるようマニュアルを作成した.

（※文責: 今井啄人）

動画

動画では, 電源の繋げ方からプログラムの立ち上げ方, プログラムの終了の仕方まで解説している. ハードウェアのセッティングから OS の立ち上げまでを説明した前編と, プログラムのセッティング, 利用方法を説明した後編を作成した.

（※文責: 今井啄人）

ドキュメント

ドキュメントでは, 動画より詳細に内容を解説している. Xeon Phi の概要から, プログラムへのコマンド, オプションの使い方を説明している. 加えて, 前期に ECMNET を翻訳した際に分かったことを書いた.

（※文責: 今井啄人）

第 5 章　まとめ

本項では, プロジェクト活動全体の評価を行っていく.

（※文責: 今井啄人）

5.1　前期活動結果

前期は主に, 楕円曲線法のアルゴリズムを理解するための知識が不足していたので, 基礎学習を行った. そして, 楕円曲線法での素因数分解プログラムのプロトタイプを作成した. $10^{306}+1$ を合成数としてこのプログラムを動かした結果, 発見された最も大きな素因数は 1575389803198166071 21(21 桁) であった.

（※文責: 今井啄人）

5.2　前期の問題点と後期への課題

前期に作成した ECM プログラムでは, ECMNET のランキングに掲載されているような巨大な素因数を見つけることは難しいと判断した. プログラムを高速化するためには, アルゴリズム, プログラムの両方で改良する必要があった. 後期は, 楕円曲線法 (ECM) をより早くするための理論を模索する理論班と, それを実装し並列化するプログラミング班に分かれて活動することにした.

（※文責: 今井啄人）

5.3　後期活動結果

後期は, 前期作成した ECM プログラムを改良するために, 理論班とプログラミング班に分かれて活動した.

理論班は楕円曲線法の計算量を削減することを目的に活動した. 具体的には直交座標系での計算量より少なくなる手法を探した. 論文を輪読し射影座標系やヤコビアン座標系を学んだ. それらの計算手法を手計算し検証した結果, 射影座標系を用いることによって加算の計算量を減らせることがわかった. 成果として楕円曲線法の加算の時間的なコストを約 1 割削減することができ, プログラムの高速化に貢献することが出来た.

プログラミング班は, 理論班からもらった射影座標系を使った計算をプログラミングした. 加えて, Xeon Phi 上で並列処理を行うために OpenMP を使用した. 報告書執筆段階では 36 桁の素因数を発見することが出来た.

（※文責: 狩野大樹）

5.4　今年度の成果

素因数分解プロジェクトは, 今回が初年度であった. 一年間の活動を通じて, 我々は以下のことを達成した.

まず一つ目に, 前期に基礎学習を通じて楕円曲線法のアルゴリズムを理解することが出来た. 二つ目に, 楕円曲線法による素因数分解プログラムを作成した. 三つ目に, 楕円曲線法のアルゴリズムの改良を行った. 四つ目に, ECM プログラムを Xeon Phi 上で並列実行を可能にした.

本プロジェクトの目標は, 楕円曲線法プログラムの高速化と, ECMNET のランキングへの挑戦であった. 前者は, 射影座標系を採用し, Xeon Phi 上での並列処理を行うことで, 最大 20% 高速化することが出来た. ECMNET のランキングに名前を載せるには, 執筆時点で 64 桁以上の素因数を発見する必要があるが, 36 桁までの素因数しか発見できなかった.

（※文責: 今井啄人）

5.5　反省点

本項では, 素因数分解プロジェクトの反省点を述べていく.

まず一つ目に, プログラムを実行する時間が十分ではなかったことが挙げられる. これは, プログラムを完成させる予定が 2 週間遅れてしまったことが原因である. プログラミング班全員がグループでプログラミングを行った経験がなかったために, どのように作業を分担し進めたらいいか分からず, コーディングが滞ってしまうことがあった. 加えて, Xeon Phi 上で並列処理を行うために OpenMP を組み込んだ際, プログラムが正しく動作せず, その調整に時間がかかってしまった. その結果, プログラムが期待通りに動くようになったのは 12 月に入ってからであった. 一般に, 大きな素因数を発見するには数ヶ月間プログラムを動かし続ける必要がある.

二つ目に, 楕円曲線法のアルゴリズムを改良する際に, 参考にした文献の量が足りなかった. また, 文献の内容が正しいことを確認するために, 文献を最初から最後まで目を通し, 内容を証明する必要があった. 理論班全員で式一つ一つを計算し, 証明するのに一ヶ月以上かかってしまった.

三つ目に, 時間の都合上, プログラムに実装できなかった手法がいくつかあった. 例えば, ヤコビ座標系を使った楕円曲線上での計算, 高速フーリエ変換, 誕生日攻撃をつかったアルゴリズム等がある. 実装することが出来れば, 更にプログラムを高速化することができたかもしれない.

（※文責: 今井啄人）

第 6 章　展望

前期には主に楕円曲線法の基礎的な知識を学び, それを元に簡易なプログラムを作成し 21 桁の素因数を見つけた. 後期ではプログラムの高速化を目標に, 手法を模索する理論班とプログラムを実装・改良するプログラミング班に分かれて活動した. 理論班はさまざまな手法を模索し, 射影座標系を用いて計算コストを 1 割削減することができた. プログラミング班は Xeon Phi 上で並列処理を行うために OpenMP ををを使用し, 報告書執筆段階で 36 桁の素因数を確認できた. しかし, 班分けの遅さや, プログラムの完成の遅れ, 実装できなかった手法の存在, 参考文献の少なさから思うようにいかず, 目的であった ECMNET のランクインを実現できなかった.

我々, 素因数分解プロジェクトが来期よりよい結果を出すためには因数分解プログラム「funecm」の改良をする必要がある. そして, 後期の早い段階から班分けを行い, 班ごとの成果を良いものにする. 理論班は射影座標系を用いたが, それだけではなくさまざまな参考文献を元に他の手法を試みて, 最善の手法を模索し, プログラミング班の手助けをする. プログラミング班は実装できなかったヤコビアン座標系の 2 倍算と高速フーリエ変換のプログラミング・実装を早い段階から行う. それによって, Xeon Phi でのプログラム実行時間を多く取り, 巨大な素因数を見つけ, ECMNET のランクインを目指す.

（※文責: 清水目佳樹）

参考文献

[1] ECMNET http://www.loria.fr/~zimmerma/records/ecmnet.html（最終アクセス 2015 年 1 月 9 日）

[2] 永井善孝（公立はこだて未来大）/伊豆哲也（富士通研）/白勢政明（公立はこだて未来大)”楕円曲線加算公式の改良”,2013

[3] ジム・ジェファース/ジェームス・レインダース, ”インテル Xeon Phi コプロッセサー ハイパフォーマンス・プログラミング”, 株式会社 カットシステム, 2014

[4] The GNU MP BIgnum Library https://gmplib.org/ (最終アクセス 2015 年 01 月 07 日)