山口大学大学院東アジア研究科
博士論文

マルチプロセッサスケジューリング問題におけるタスクグラフのブロック化に関する研究

2015年8月31日

陳　霖

学位論文内容要旨

学位論文題目：マルチプロセッサスケジューリング問題におけるタスクグラフのブロック化に関する研究

指導教員　　：葛　崎偉　教授

申請者名　　：陳　霖　（チン　リン）

平成24年度入学
山口大学大学院　東アジア研究科　博士後期課程

マルチプロセッサによる分散処理は，システムの実行時間を短縮する有効な手段の一つであり，計算機システムにとどまらず，車両制御システムを含むさまざまなシステムにも利用されている．現在，マルチコアプロセッサを含め，マルチプロセッサシステム用の並列プログラミングは難易度が高く，所望の性能を得るために長期間の並列チューニングが必要であることが知られており，並列ソフトウェア生産性向上が大きな課題となっている．

マルチプロセッサシステムにおいて，その処理能力を最大限に引き出すためには，プログラムの実行に関する最適なスケジュールが必要であり，そのスケジュールを導き出すスケジューリング手法が重要となる．一般的にマルチプロセッサスケジューリング問題は，プログラムを表すタスクグラフとプロセッサ数が与えられた場合に，タスクグラフの実行時間が最小となるように，プロセッサに割り当てられるタスクの実行順序を決定することである．しかしながら，この問題は極めて難しく，以下の特別な２つの場合を除いて，NP-困難である．

(1) タスクグラフのすべてのノードが同じ実行時間を持つ根付き有向グラフであり，プロセッサの数は任意であるがその処理能力は同じである場合.

(2) タスクグラフのすべてのノードが同じ実行時間を持ち，プロセッサの数は２である場合.

このような状況の下では，プログラムの実行時間が最小にならなくとも，最適に近い，つまり準最適なスケジュールを求める手法が必要となる．これに関しては，これまでにもさまざまな手法が考えられているが，本研究の主要な動機は通信時間の増加を伴わずタスクグラフにブロック化することにより，スケジュール結果を短縮することである．ブロック化とは，ある条件を満たした複数のノードを 1 つのブロック (新たなノード) に結合することである．ブロック化することによって，これらのノードを実行する際にノード間の通信を行う必要がなくなる．新たなノードの実行時間はブロックに含まれるすべてのノードの実行時間の総和である．なお，これらのノードを 1 つのブロックに結合しない場合はノード間の通信時間が必要である．本研究では，プロセッサ間のデータの通信に時間的コストを要するものとし，通信時間の増加を伴わず，スケジュールをより短くするようなブロック化法と再ブロック化法を提案することを目的としている．

本論文は以下のように第 1 章から第 6 章まで構成される．

第 1 章では研究背景と研究目的を述べている. マルチプロセッサによる分散処理は，システムの実行時間を短縮するための有効な手段であり，様々な分野で利用されている．マルチプロセッサシステムを最大限に活用するために，プログラムの実行に関する最適なスケジューリング方法の開発が必要である．この研究はプログラムの実行時間をより短くするためのブロック化手法の提案を目的としている．

第 2 章では，対象のタスクグラフが有向閉路のない DAG であり，各ノード間に先行制約があることを説明している．通信時間を要するマルチプロセッサシステムのモデルを定義する．さらに，タスクグラフにおけるブロックの概念を定義している．

第 3 章と第 4 章では，それぞれブロック化法と再ブロック化法を提案している．ブロック化法は，4 種類の親子ノード集合のいずれかをもつ連結部分グラフを一つにする基本ブロック化法と 4 つの構造パターンのいずれかをもつ連結部分グラフを一つにする追加ブロック化法からなっている．ブロック化法では，基本ブロック化法を先に適用し，次に追加ブロック化法を適用するように実行している．再ブロック化法は，スケジューリングの結果に基づいて行う．まず，ノードの実行に関する最早と最遅の開始時刻を用いた LST-EST スケジューリング法を提案し，このスケジューリングの実行結果に基づいて，一つのプロセッサで実行可能な連結部分グラフを一つにする．

第 5 章では，提案したブロック化法と再ブロック化法を評価するためのシミュレーシ

ョン方法およびその実験結果について述べている．実験用のタスクグラフはMATLABを用いて作成した自動車制御用のSimulinkモデルから抽出して利用している．提案したブロック化法と再ブロック化法を適用した結果，通信時間が50%以上短縮されたことを確認しており，提案手法が自動車制御用システムの実行に有効であることを明らかにしている．

第6章では，まずこの論文で提案されたブロック化法と再ブロック化法，また実用システムに対する提案方法の有効性についてまとめている．次に，本研究で得られる成果に基づいてエネルギー伝送を含む社会システムにおける諸問題を解決することについて言及している．最後に，今後の研究課題を示している．

謝辞

本研究の遂行並びに博士論文の作成にあたり，多くのご支援とご指導を賜りました．

研究活動全般に御指導と御高配を賜りました山口大学大学院東アジア研究科葛崎偉教授に甚大なる謝意を表します．時に応じて，厳しくご指導いただいたこと，またやさしく励ましてくださったことを通して，私自身の至らなさを実感することができたことは今後の努力の糧になるものであります．私は，自分が博士課程修了後も研究者として歩む選択をした際に，誰よりも喜んでくださった先生の姿を一生忘れないでしょう．葛研究室での経験を心の糧に，今後も研究者として人の役に立っていきたいと思います．

本論文の作成に当たり，副指導教員として山口大学大学院東アジア研究科成富敬教授から多分野に渡り有益な助言を頂き，日本語を丁寧に直して頂いた御陰で，簡潔かつ適切な言葉で論文を仕上げることができました．また，副指導教員である山口大学大学院東アジア研究科福田隆眞教授から日頃より多くのご助言や励ましの言葉を頂きました．ここで，深く感謝いたします．

学位論文審査において，貴重なご指導とご助言を頂いた山口大学大学院理工学研究科山口真悟准教授に心より感謝いたします．本研究を進めるにあたり，多くのご支援とご指導を頂いた山口大学教育学部教育研究科中田充教授に深く感謝いたします．

本研究を遂行するにあたり，数々のご助言やご指導をいただき，また現場調査やデータ提供等においてもご協力頂いた富士通テン株式会社の斗納宏敏氏，山崎剛氏，神山尚也氏，嶋田一行氏には，深く感謝いたします．

研究室の皆様にも，日頃より暖かくご支援いただき，また学位論文を作成する際には日本語のチェックをしていただきました．誠にありがとうございました．

最後になりましたが，博士課程に進学する機会を与えてくださり，ありとあらゆる場面で私を温かく見守り続けてくれた両親に深く深く感謝いたします．これからすこしずつ時間をかけて恩返しをしていきたいと思います．

表記法/Notation

$x \in X$:	x は X 集合の要素である.
$x \notin X$	x は X 集合の要素ではない.
$X \cup Y$:	X と Y の和集合である.
$X \cap Y$:	X と Y の交差集合である.
$\sum x_i$:	x_i すべての要素の合計である.
ϕ:	空集合である.
$\|X\|$:	X 集合の要素数である.
$\max X$:	X 集合の最大要素.
$\min X$:	X 集合の最小要素.
$X \leftarrow \{x_1, x_2, \cdots, x_n\}$:	X 集合に $x_1, x_2, \cdots, x_n$ の要素を追加である.
$L \leftarrow t_1 t_2 \cdots t_n$:	L の左からの順に子供 $t_1, t_2, \cdots, t_n$ を追加である.

目次

論文概要 **3**

謝辞 **5**

表記法 **7**

目次 **9**

図目次 **1**

表目次 **8**

1 はじめに **11**

1.1 研究の背景と動機 11

1.2 論文の構成 15

2 本研究の各定義 **17**

2.1 タスクグラフの定義 17

2.2 モデルの定義 19

2.3 ブロックの定義 19

3 ブロック化法の提案 **23**

3.1 基本ブロック化法の提案 23
3.2 追加ブロック化法の提案 28
3.2.1 最早開始時刻と最遅開始時刻 28
3.2.2 追加ブロック化法 30

4 再ブロック化法の提案 37

4.1 最早最遅開始時刻に基づいたスケジューリング法 37
4.2 再ブロック化法 41

5 シミュレーション結果と考察 45

5.1 Matlab を用いた Simulink モデルの隣接情報抽出 45
5.1.1 タスクグラフの行列について 46
5.1.2 閉路と多重辺の処理について 46
5.1.3 サブシステムブロックの展開について処理 49
5.1.4 Matlab を用いた Simulink モデルの隣接情報抽出するプログラフの紹介 54
5.2 提案した方法のシミュレーションの実験と評価 57
5.3 Matlab プログラミング言語に基づくブロック化の実現 59
5.3.1 coolant タスクグラフについて 60
5.3.2 データを読み込むサブプログラム (Chen.m) 62
5.3.3 各ノードに最早最遅開始時刻と優先リストを求めるサブプログラム (List.m) 63
5.3.4 クリティカルパス法（CP 法）というシミュレーション法を求めるプログラム (CP.m) 64

5.3.5 各ノードの先祖と子孫を求めるサブプログラム (Predecessor.m, Successor.m) 65
5.3.6 ブロック化法の基本ブロック化法のサブプログラム (Block.m) 66
5.3.7 ブロック化法の追加ブロック化法のサブプログラム (Newblock.m) 68
5.3.8 ブロック化前後の対応表を求めるサブプログラム（blocktable.m） 71
5.3.9 評価用スケジューリング法のプログラム（Scheduling_1.m） . 72
5.3.10 再ブロック化法の専用スケジューリング法 (LST-EST 法) のサブプログラム（Scheduling.m） 74
5.3.11 再ブロック化法のサブプログラム（ReBlock.m） 75

6 おわりに 77

6.1 研究の成果 77
6.2 本研究の展望 79
6.3 今後の課題 81

参考文献 83

付録 90

図目次

1.1 現在のマルチモデル構築 13
1.2 1 つの Simulink モデル内のマルチモデル構築 13

2.1 タスクグラフ G 18
2.2 データの書き出しと読み込み 20
2.3 タスクグラフ G 中のブロック B 20

3.1 基本ブロック化法の 4 つのパターン 24
3.2 基本ブロック化法の操作 O1 25
3.3 基本ブロック化法の操作 O2 25
3.4 基本ブロック化法の操作 O3 26
3.5 基本ブロック化法の操作 O4 26
3.6 基本ブロック化法の実行結果 27
3.7 ノード z_x とその親子ノード 29
3.8 親子関係の 4 つのパターン 31
3.9 追加ブロック化法にブロックできる場合 31
3.10 追加ブロック化法のパターン (1) 32
3.11 追加ブロック化法のパターン (2) 32
3.12 追加ブロック化法のパターン (3) 33

3.13 追加ブロック化法のパターン (4) 33
3.14 追加ブロック化法の実行結果 35

4.1 タスクグラフの優先リスト 38
4.2 各ノードの選択範囲 39
4.3 ノードの選択範囲を使う LST-EST 法 40
4.4 ノードの選択範囲を使わずに普通のスケジューリング法 40
4.5 元のタスクグラフにブロック化法の実行結果 41
4.6 実行されたタスクグラフのスケジュール結果 41
4.7 実行されたタスクグラフのスケジュール結果 42
4.8 ノード z と z' がブロック B に構成 42
4.9 ブロック B は単一のノードに縮約 42
4.10 LST-EST 法に基づいてタスクグラフ G' のスケジュール結果 43
4.11 再ブロック化法の結果 43
4.12 親子関係のノードと別のパス 43

5.1 図 2.1 のタスクグラフの行列の形式 47
5.2 閉路の処理 48
5.3 多重辺の処理 49
5.4 各ブロックの親情報を取得する処理 50
5.5 サブシステム内の Inport と Outport 51
5.6 サブシステムブロックの展開の手順 1 52
5.7 サブシステムブロックの展開の手順 2 52
5.8 サブシステムブロックの展開の手順 3 53
5.9 サブシステムブロックの展開の手順 4 53
5.10 サブシステムブロックの展開の手順 5 54

5.11 実行用プログラムの名前と保存場所 55
5.12 プログラムの実行画面 55
5.13 システム model1 中のサブシステムブロック Coolant etc 56
5.14 サブシステムブロック Coolant etc 中のサブシステムブロック CAL_O2 56
5.15 サブシステムブロック sample7 57
5.16 sample7 のタスクグラフの行列 58
5.17 sample7 のタスクグラフの対応表 58
5.18 coolant タスクグラフの txt ファイル 60
5.19 coolant タスクグラフ 60
5.20 coolant タスクグラフの行列 61
5.21 ファイルからの読み込み 62
5.22 ノード z_2 とタスクグラフの構造体 62
5.23 最早最遅開始時刻と優先リストの情報を追加したノード z_2 とタスクグラフの構造体 63
5.24 ブロック化法で実行したタスクグラフ前後の最長パスの長さ 64
5.25 先祖の情報を追加したノード z_2 の構造体 65
5.26 子孫の情報を追加したノード z_2 の構造体 65
5.27 基本ブロック化法（1）: ブロック化できる親子関係のノード 66
5.28 基本ブロック化法（2）: 重複した親子関係のノードの組み合わせ . . . 66
5.29 基本ブロック化法で実行して得られたタスクグラフの行列 67
5.30 追加ブロック化法 1 回目と 2 回目の実行結果 68
5.31 追加ブロック化法最後の 1 回目の実行結果 69
5.32 New_Graph.txt ファイル 69
5.33 追加ブロック化法最後の実行結果 70
5.34 ブロック化前後の対応結果 71

5.35 ブロック化法で実行前後のタスクグラフのスケジュール結果 72
5.36 再ブロック化法で実行前後のタスクグラフのスケジュール結果 74
5.37 再ブロック化法で実行した結果 75
5.38 再ブロック化法で実行して得られたタスクグラフ 76

6.1 世界の一次エネルギー消費量の推移 [56] 79
6.2 中国の鉱物の分布 [57] 80
6.3 地域間連系線 [63] 81
6.4 車両制御システムモデル model1 91
6.5 Coolant Control サブシステムモデル model1/Coolant etc 92
6.6 CAL_O2 サブシステムモデル model1/Coolant etc/CAL_O2 92
6.7 Engine Output Control サブシステムモデル model1/Engine output . 93
6.8 flag_separater サブシステムモデル model1/Engine output/flag_separater 94
6.9 judge cylinder number サブシステムモデル model1/Engine output/ flag_separater/judge cylinder number 94
6.10 cylinder 1 サブシステムモデル model1/Engine output/cylinder 1 . . . 95
6.11 cylinder 2 サブシステムモデル model1/Engine output/cylinder 2 . . . 95
6.12 cylinder 3 サブシステムモデル model1/Engine output/cylinder 3 . . . 96
6.13 cylinder 4 サブシステムモデル model1/Engine output/cylinder 4 . . . 96
6.14 Vehicle Control サブシステムモデル model1/Vehicle 97
6.15 Fuel Behavior Control サブシステムモデル model1/Fuel behavior . . . 98
6.16 flag_separater サブシステムモデル model1/Fuel behavior/flag_separater 98
6.17 cylinder 1 サブシステムモデル model1/Fuel behavior/cylinder 1 . . . 99
6.18 cylinder 2 サブシステムモデル model1/Fuel behavior/cylinder 2 . . . 99

6.19 cylinder 3 サブシステムモデル model1/Fuel behavior/cylinder 3 . . . 100
6.20 cylinder 4 サブシステムモデル model1/Fuel behavior/cylinder 4 . . . 100
6.21 Induction Control サブシステムモデル model1/Induction System . . . 101
6.22 eata サブシステムモデル model1/Induction System/eata 101
6.23 abstract.m（１） 102
6.24 abstract.m（２） 103
6.25 abstract.m（３） 104
6.26 abstract.m（４） 105
6.27 abstract.m（５） 106
6.28 abstract.m（６） 107
6.29 abstract.m（７） 108
6.30 abstract.m（８） 109
6.31 abstract.m（９） 110
6.32 block_construction.m（１） 111
6.33 block_construction.m（２） 112
6.34 block_construction.m（３） 113
6.35 block_construction.m（４） 114
6.36 block_construction.m（５） 115
6.37 block_construction.m（６） 116
6.38 block_construction.m（７） 117
6.39 block_construction.m（８） 118
6.40 block_construction.m（９） 119
6.41 block_construction.m（１０） 120
6.42 block_construction.m（１１） 121

6.43 block_construction.m（１２） 122
6.44 block_construction.m（１３） 123
6.45 block_construction.m（１４） 124
6.46 block_construction.m（１５） 125
6.47 block_construction.m（１６） 126
6.48 block_construction.m（１７） 127
6.49 block_construction.m（１８） 128
6.50 block_construction.m（１９） 129
6.51 block_construction.m（２０） 130
6.52 block_construction.m（２１） 131
6.53 block_construction.m（２２） 132
6.54 block_construction.m（２３） 133
6.55 block_construction.m（２４） 134
6.56 block_construction.m（２５） 135
6.57 block_construction.m（２６） 136
6.58 block_construction.m（２７） 137
6.59 block_construction.m（２８） 138
6.60 block_construction.m（２９） 139
6.61 block_construction.m（３０） 140
6.62 block_construction.m（３１） 141
6.63 block_construction.m（３２） 142
6.64 block_construction.m（３３） 143
6.65 block_construction.m（３４） 144
6.66 block_construction.m（３５） 145
6.67 block_reconstruction.m（１） 146

6.68 block_reconstruction.m（２） 147
6.69 block_reconstruction.m（３） 148
6.70 block_reconstruction.m（４） 149
6.71 block_reconstruction.m（５） 150
6.72 block_reconstruction.m（６） 151
6.73 block_reconstruction.m（７） 152
6.74 block_reconstruction.m（８） 153
6.75 block_reconstruction.m（９） 154
6.76 block_reconstruction.m（１０） 155
6.77 block_reconstruction.m（１１） 156
6.78 block_reconstruction.m（１２） 157
6.79 block_reconstruction.m（１３） 158
6.80 block_reconstruction.m（１４） 159
6.81 block_reconstruction.m（１５） 160
6.82 block_reconstruction.m（１６） 161
6.83 block_reconstruction.m（１７） 162
6.84 block_reconstruction.m（１８） 163
6.85 block_reconstruction.m（１９） 164
6.86 block_reconstruction.m（２０） 165
6.87 Scheduling_1.m（１） 166
6.88 Scheduling_1.m（２） 167
6.89 Scheduling_1.m（３） 168
6.90 CP.m 169

表目次

5.1 実システムにブロック化法と再ブロック化法に基づいてスケジュール結果 59

はじめに

1.1 研究の背景と動機

マルチプロセッサシステムについて，単一のコンピュータシステム中には複数の中央処理装置（CPU）が存在する．複数の CPU はコンピュータのバスを利用して各メモリおよびその他の周辺機器を繋げて，機器が互いにコミュニケーションできる [1]．マルチプロセッサシステムは 2 つの種類があり，密結合マルチプロセッサシステムと疎結合マルチプロセッサシステムである．密結合マルチプロセッサシステムは 1 つの OS が複数の CPU を制御し，それぞれの CPU は主記憶装置を共有するシステムであり，このシステムの特徴は処理の効率向上である [2]．疎結合マルチプロセッサシステムはそれぞれの処理装置を独立した OS が制御し，主記憶装置も独立して持っているシステムであり，このシステムの特徴は信頼性が高いである [2]．本研究はシステムの全体の処理能力が高くため，密結合マルチプロセッサシステムを利用することになった．

マルチプロセッサシステムは，情報処理，ロボットの制御，動的システムの高速シミュレーション，車両制御システムなどさまざまな計算機システムに利用されている [3, 4, 5, 6]．現在，マルチコアプロセッサを含め，マルチプロセッサシステム用の並列プログラミングは難易度が高く，所望の性能を得るために長期間の並列チューニングが必要であることが知られており，並列ソフトウェア生産性向上が大きな課題となっている [7, 8, 9, 10]．

近年，各大学において，PC やネットワークという情報機器を利用して様々な教育用システムが構築されている [11, 12, 13]．例えば，学生ユーザ管理におけるユーザ

アカウントデータベースの統合管理システム [14, 15], 学部教育における情報教育用PC クラスタシステム [16, 17, 18] などである．計算機の性能は年々高くなっているとともに，教育用システムの実行スピードを進化させる必要がある．現在，複数のコアを持つ計算機は幅広く利用されている．それらは現在マルチシングルコアプロセッサシステムによって構築されているが，マルチコアプロセッサシステムを用いることで 1 つの問題を複数個のコアで並列処理を行うことで高速に処理することができる [19]．マルチコアプロセッサシステムを用いれば教育用システムの効率化が図れる．

マルチプロセッサシステムにおいて，その処理能力を最大限に引き出すためには，プログラムの実行に関する最適なスケジュール結果が必要であり [20, 21, 22]，そのスケジュール結果を導き出すスケジューリング手法が重要となる．一般的にマルチプロセッサスケジューリング問題は，プログラムを表すタスクグラフ (即ち有向そして閉路がないグラフ，DAG) とプロセッサ数が与えられた場合に，タスクグラフの実行時間が最小となるように，プロセッサに割り当てられるタスク (以後にノードと呼ぶ) の実行順序を決定することである．しかしながら，この問題は極めて難しく，以下の特別な 2 つの場合を除いて，NP-困難である [23, 24, 25, 26].

1 タスクグラフのすべてのノードが同じ実行時間を持つ根付き有向グラフであり，プロセッサの数は任意であるがその処理能力は同じである場合 [27].
2 タスクグラフのすべてのノードが同じ実行時間を持ち，プロセッサの数は 2 である場合 [28].

本研究の主要な動機は通信時間の増加を伴わずタスクグラフにブロック化することにより，スケジュール結果を短縮することである．ブロック化とは，ある条件を満たした複数のノードを 1 つのブロック (新たなノード) に結合することであり，これによって，これらのノードを実行する際にノード間の通信を行う必要がなくなる．新たなノードの実行時間はブロックに含まれるすべてのノードの実行時間の総和である．なお，これらのノードがブロックに結合していない場合はノードとノード間の通信時間が必要である．

MATLAB/Simulink に基づいてリアルタイムシステムの開発の問題 [29] に対してブロック化のことを利用して説明する．それはマルチコア並列システム中で複数の

Simulink モデルを*.mdl として個別に登録する必要がある．しかし，図 1.1 に示すように 1 つの大規模なモデルを個々の*.mdl として分割するのは非常に困難であり開発効率が悪い [29]．そこで，マルチコア並列システムに適した大規模なモデルを開発することが容易となるために図 1.2 に示すように，サブシステムにいくつかの単位モデルを組み合わせるスケジューラ機能が望ましい．

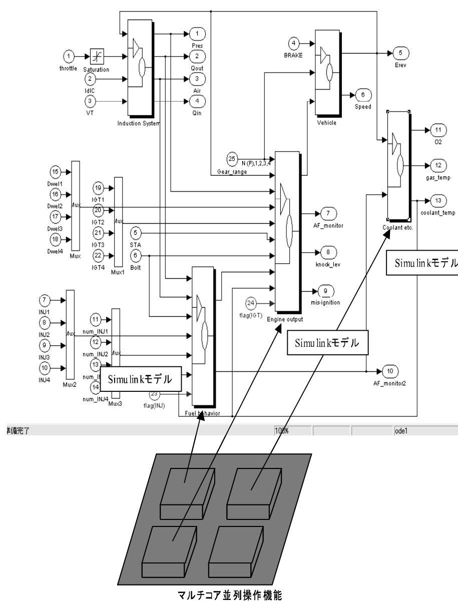


図 1.1: 現在のマルチモデル構築

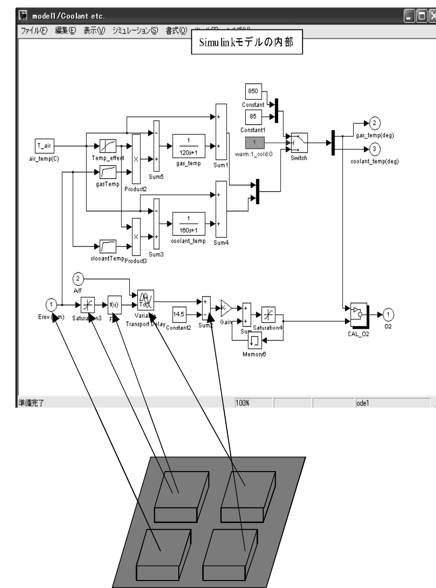


図 1.2: 1 つの Simulink モデル内のマルチモデル構築

マルチプロセッサスケジューリング問題はとても難しい問題である．このような状況の下では，最適なスケジュール結果を求めるスケジューリング法が必要となる．ですが，すでに世の中でさまざまなスケジューリング法がある．例えば，クリティカルパスに基づいたスケジューリング法 [30, 31]，最早開始時刻と最遅開始時刻に基づいたスケジューリング法 [32, 33] など．本論文ではタスクグラフのスケジュール結果を短縮できるブロック化法を提案して，ブロック化法で実行されたタスクグラフのスケジュール結果をさらに短縮するためにスケジューリング技術とノードの接続関係を利用してその実行されたタスクグラフに再びブロック化する再ブロック化法を提案する．ここで，ブロック化に対していくつかの類似する研究がある．それはマルチ制約グラフ分割に基づいたノードの割り当ての研究 [34]，CASS-II のアル

ゴリズムに基づいたノードのクラスタリングの研究である [35]．これらの研究は両方ともノード間の通信時間が減少し，スケジュール結果を短縮させることができる．しかし，これらの研究は我々の研究の前提条件を満たしていない．その前提条件はブロックの実行に必要な全ての入力データはブロックの実行前に読み込まれるということと，全ての出力データは実行後に書き出されることである．したがって，これらの提案されている方法は我々の問題に適用することはできない．

1.2 論文の構成

本論文は以下のように構成される:

1 章では本論文の研究背景を紹介し，そして自分の研究目的を述べる.

2 章では，タスクグラフ，ノード間の先行制約，通信時間を要するマルチプロセッサシステムのモデル，タスクグラフのブロック，各定義を述べる．

3 章では，ブロック化法を提案する．ブロック化法は基本ブロック化法と追加ブロック化法を含む．まず基本ブロック化法の 4 つパターンを説明し，次に最早最遅開始時刻を説明し，最後追加ブロック化法の 4 つのパターンを説明する.

4 章では再ブロック化法を提案する．まず最早最遅開始時刻によってタスクグラフのスケジュール結果を求める LST-EST 法を説明し，次にそのスケジュール結果に基づいて再ブロック化法を提案する．

5 章では, 提案したブロック化法と再ブロック化法を評価するためのシミュレーション実現について述べる．まず，Matlab を用いた Simulink モデルの隣接情報抽出し，タスクグラフの各情報を求める．次は得られたタスクグラフを利用し，提案した方法でシミュレーション実験を行う．提案したブロック化法と再ブロック化法の有効性を確認する．最後では，シミュレーション用の各実行用プログラムを紹介し，実行結果を出す.

6 章では本論文で得られた結果をまとめ，今後の課題について述べる.

付録では実験用システムモデルの各サブシステムを展開し，システムの構造を示す．または Matlab で作成された各スクリプト M-ファイル (実行用プログラム) を紹介する．

第 2 章

本研究の各定義

2 章では，タスクグラフ，ノード間の先行制約，通信時間を要するマルチプロセッサシステムのモデル，タスクグラフのブロック，各定義を述べる．

2.1 タスクグラフの定義

[**定義 1**] $G=(N,E)$ をタスクグラフとする．G はソースノード s とシンクノード t を持つ DAG(Directed Acyclic Graph)[36, 37]．そして，z_i と z_j は G のノードとする．

- 各ノード $z \in N$ に対して，s から z への有向パス [38] が存在し，ノードを表す．各有向枝 $e \in E$ は 2 つのノード間の接続を表す．
- 有向枝 [39]$e=(z_i, z_j)$ は，そのノード間の先行制約 [40] と通信伝送路を表す．ノード z_i と z_j を結ぶ有向枝 e が存在した時，ノード z_j はノード z_i の実行が終わらない限り，実行を開始できない．
- 有向枝 $e=(z_i, z_j)$ は重み $c(z_i, z_j)$(または c_e) を持ち，z_i と z_j が異なるプロセッサで処理された時に要する通信時間を表す．また，任意の z_i に対し，$c(s, z_i)$ と $c(z_i, t)$ は 0 とする．
- 各ノード z_i は実行時間 $\gamma_i(\geq 0)$ を持ち，その実行は中断できない．特にソースノード s とシンクノード t の実行時間は 0 とする．

□

図 2.1 タスクグラフの例を示している．図中の丸の中の数字はソースノードとシンクノードそしてノードの番号を表す．括弧の中の数字は各ノードの実行時間を表す．また，矢印の横の数字は通信時間 (書き出し時間と読み込み時間) を表す．例え

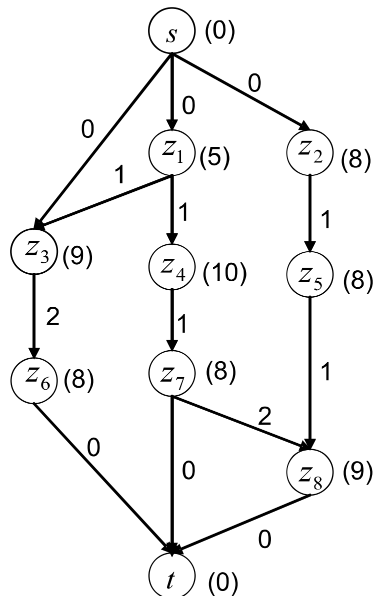


図 2.1: タスクグラフ G

ば，ノード z_2 の実行時間が 8，読み込み時間が 0，書き出し時間が 1 である．

[**定義 2**]z_i と z_j は G 中の 2 つのノードとする．

- z_i から z_j への有向パスが存在するとき，z_i を z_j の先祖 (Predecessor)，z_j を z_i の子孫 (Successor) と呼ぶ，z の先祖と子孫の集合をそれぞれ $Pre(z)$，$Suc(z)$ と表記する．
- $(z_i, z_j) \in E$ である時，z_i を z_j の親 (Immediate Predecessor)，
  z_j を z_i の子 (Immediate Successor) と呼ぶ，z の親と子の集合をそれぞれ $IP(z)$，$IS(z)$ と表記する．
- ソースノード s からシンクノード t までの最長パスをクリティカルパス [41] と呼び，(CP) と表記する．また，パスの長さはパスに含まれるすべてのノードの実行時間とすべての入力枝の重みとすべての出力枝の重みの合計である．

□

2.2 モデルの定義

本論文で対象とするマルチプロセッサシステムのモデルを以下のように定義する．

[**定義** 3] 以下のすべてを満たすものをマルチプロセッサシステムモデルとする．

- 全プロセッサ $p_k \in PE$ の処理能力が等しく，PE がマルチプロセッサシステムで使用されるプロセッサの集合である．
- 通信時間は異なるプロセッサが 2 つの隣接ノードを実行する時に，共有メモリへデータの書き出しと共有メモリからデートの読み込みに要する時間である．通信時間中の読み込み時間と書き出し時間は異なるプロセッサ間で定数であるが，1 つのプロセッサ中で 0 である [42].
- プロセッサの p_k はすべての入力からデータを読み込むとノードの実行を開始し，すべての出力へのデータを書き出すと，実行を終了する．
- ブロックに含まれるすべてのタスクが同一のプロセッサで実行する．ブロックの実行に必要な全ての入力データはブロックの実行前に読み込まれる．さらに，全ての出力データは実行後に書き出される．

□

データの書き出しと読み込みを図 2.2 を用いて説明する．図 2.2 中の実線①はデータの書き出しを表し，プロセッサで処理したタスクの出力データを自身のローカルメモリに書き出し，書き出し時間は 0 である．また，実線②はデータの読み込しを表す．実線②は p_1 が処理したタスクの出力データが自身のローカルメモリから読み込まれることを表し，読み込み時間も 0 である．実線③，④は p_2 が処理したタスクの出力データが p_1 が取る場合，共有メモリを通して読み込みと書き出しのことを表し，その読み込みと書き出しに時間がかかる．

2.3 ブロックの定義

[**定義** 4] タスクグラフ $G{=}(N, E)$ の部分グラフの B は以下の条件を満たした，G の縮約可能なブロックと呼ぶ (図 2.3).

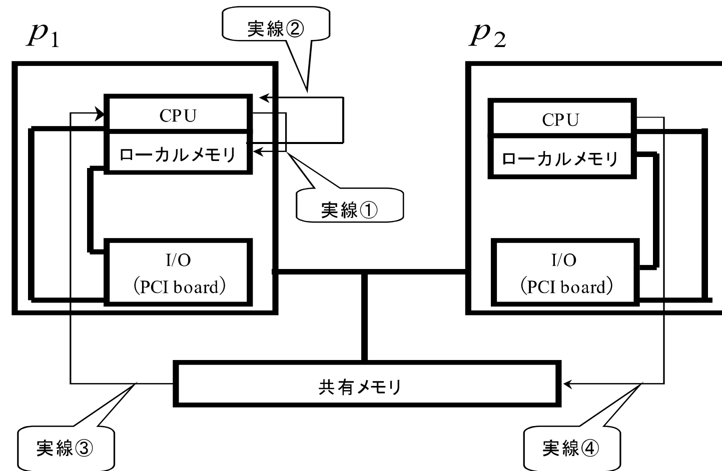


図 2.2: データの書き出しと読み込み

- B は連結的な部分グラフ.
- B に含まれるノードが単一のノードに縮約した場合，結果として得られたグラフが DAG(Directed Acyclic Graph) である.

□

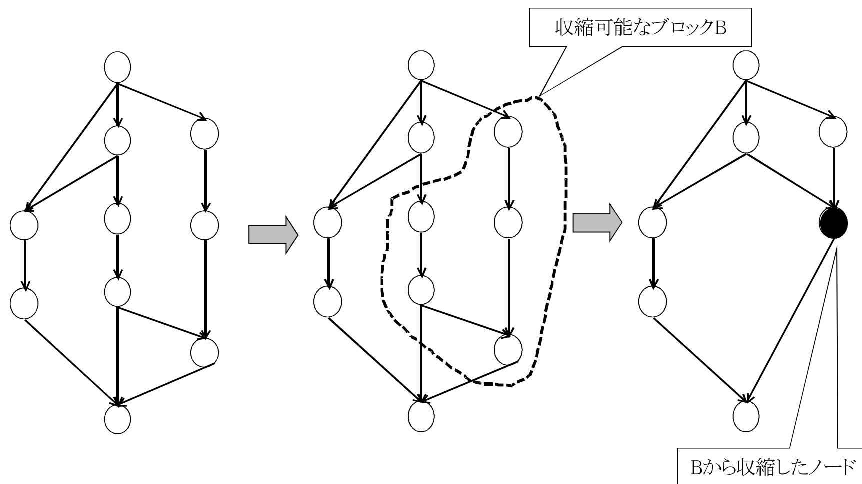


図 2.3: タスクグラフ G 中のブロック B

[**定義 5**]$G = (V, E)$ と $H = (V', E')$ はタスクグラフである．以下の条件を満たした時，G と H は等価であるという.

- $H = (V', E')$ は G のいくつかの部分グラフをそれぞれ一つのノードに縮約し，縮約された各ノードの実行時間は部分グラフの実行時間の総和であるように操作して得られたグラフである．
- G における 2 つのノード，$u, v \in V \cap V'$ について，u が v の先祖ならば H においても u が v の先祖である．また，その逆も成り立つ．

□

第 3 章

ブロック化法の提案

本章では，ブロック化法を提案する．ブロック化法は基本ブロック化法と追加ブロック化法を含む．まず基本ブロック化法の 4 つパターンを説明し，次に最早最遅開始時刻を説明し，最後追加ブロック化法の 4 つのパターンを説明する．

3.1　基本ブロック化法の提案

この節では，定義 3 によってマルチプロセッサシステムで実行されるタスクグラフの通信時間減少させるために，基本ブロック化法を提案する．

次の定義は，ノードのパターンを与える．また，それらのパターンは定義 4 の条件を満たす部分グラフである．即ち，与えるパターンはブロックである．

[**定義 6**] G=(N,E) はタスクグラフとする．そして，(z_i,z_j)∈E は G の有向辺 [43] とする．

- $|IP(z_j)|$=1 かつ $|IS(z_i)|$=1 を満たす z_i，z_j を極小親子と呼び，$\{z_i,z_j\}$ を極小親子集合 (Minimal Parent-Child Set) という．MPC で表示する．
- $Pre(z_j)$=$Pre(z_i)$∪$\{z_i\}$ かつ $|IS(z_i)|$=1 を満たす z_i，z_j を同先祖親子と呼び，$\{z_i,z_j\}$ を同先祖親子集合 (Set of Parent-Child Nodes with Same Predecessors) という．PCP で表示する．
- $Suc(z_i)$=$Suc(z_j)$∪$\{z_j\}$ かつ $|IP(z_j)|$=1 を満たす z_i，z_j を同子孫親子と呼び，$\{z_i,z_j\}$ を同子孫親子集合 (Set of Parent-Child Nodes with Same Successors) という．PCS で表示する．
- $Suc(z_i)$=$Suc(z_j)$∪$\{z_j\}$ かつ $(Pre(z_j)$=$Pre(z_i)\cup z_i)$ を満たす z_i，z_j を同先祖

子孫親子と呼び，$\{z_i, z_j\}$ を同先祖子孫親子集合
(Set of Parent-Child Nodes with Same Predecessors and Successors) という．
PCPS で表示する． □

図 3.1 は MPC，PCP，PCS，PCPS のノードパターンを示している．

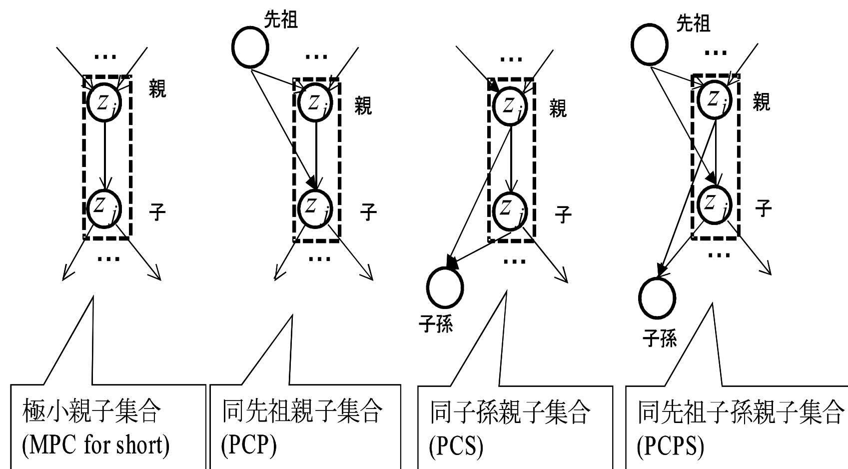


図 3.1: 基本ブロック化法の 4 つのパターン

[**補題 1**] MPC，PCP，PCS，PCPS に含まれるノードによって誘導される部分グラフは縮約可能なブロックである． □

証明: ノード z は z' の親であり，2 つのノードが同じブロック B に結合することとする．ブロック B は単一のノードに変更する時にタスクグラフの中に有向閉路ができた場合は有向枝 (z, z') 以外に，z から z' までのパスが存在することになる．上記の 4 つのノード集合の条件においては，このようなパスが存在しない．よって，ブロック B は単一のノードに変更する時にタスクグラフの中に有向閉路が存在しない．したがって，この補題が成り立つ． **Q.E.D.**

次の**操作 O1-O4** が基本ブロック化法を示している．

[**操作 O1**] 一連の MPC，$\{z_1, z_2\}, \{z_2, z_3\}, \cdots, \{z_{k-1}, z_k\}$ が存在する時に，これらのノード $z_1, z_2, \cdots, z_k$ を 1 つのブロックにする．そして，ブロックが単一のノードに縮約する (図 3.2)．ちなみにこの単一のノードの実行時間は $z_1, z_2, \cdots, z_k$ の実行時間の合計である． □

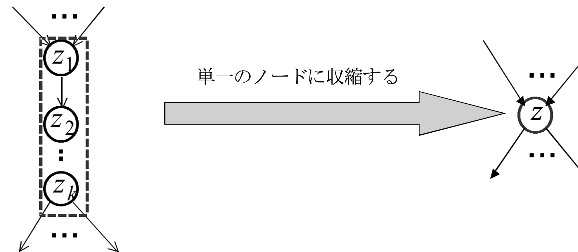


図 3.2: 基本ブロック化法の操作 O1

[**操作 O2**] 一連の PCP，$\{z_1, z_2\}$, $\{z_2, z_3\}$, $\cdots$, $\{z_{k-1}, z_k\}$ が存在する時に，これらのノード $z_1, z_2, \cdots, z_k$ を 1 つのブロックにする．そして，ブロックが単一のノードに縮約する (図 3.3)．ちなみにこの単一のノードの実行時間は $z_1, z_2, \cdots, z_k$ の実行時間の合計である．さらに，複数の有向辺が単一の有向辺に変換する． □

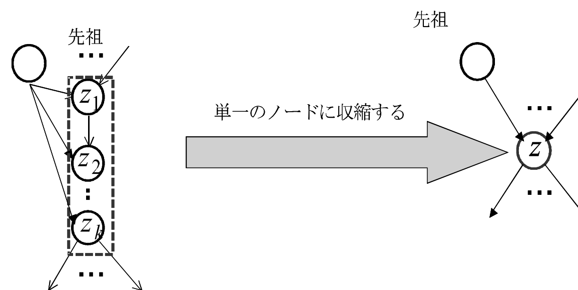


図 3.3: 基本ブロック化法の操作 O2

[**操作 O3**] 一連の PCS，$\{z_1, z_2\}$, $\{z_2, z_3\}$, $\cdots$, $\{z_{k-1}, z_k\}$ が存在する時に，これらのノード $z_1, z_2, \cdots, z_k$ を 1 つのブロックにする．そして，ブロックが単一のノードに縮約する (図 3.4)．ちなみにこの単一のノードの実行時間は $z_1, z_2, \cdots, z_k$ の実行時間の合計である．さらに，複数の有向辺が単一の有向辺に変換する． □

[**操作 O4**] 一連の PCPS，$\{z_1, z_2\}$, $\{z_2, z_3\}$, $\cdots$, $\{z_{k-1}, z_k\}$ が存在する時に，これらのノード $z_1, z_2, \cdots, z_k$ を 1 つのブロックにする．そして，ブロックが単一のノードに縮約する (図 3.5)．ちなみにこの単一のノードの実行時間は $z_1, z_2, \cdots, z_k$ の実行時間の合計である．さらに，複数の有向辺が単一の有向辺に変換する． □

[**補題 2**] G はタスクグラフ，G' は G に**操作 O1-O4** を実行して得られたグラフとす

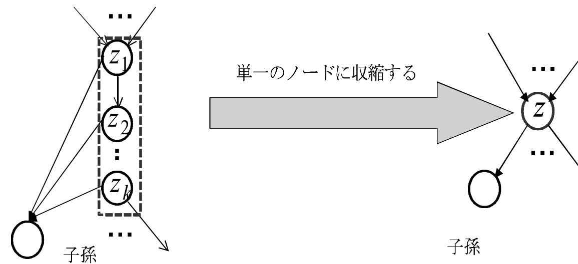


図 3.4: 基本ブロック化法の操作 O3

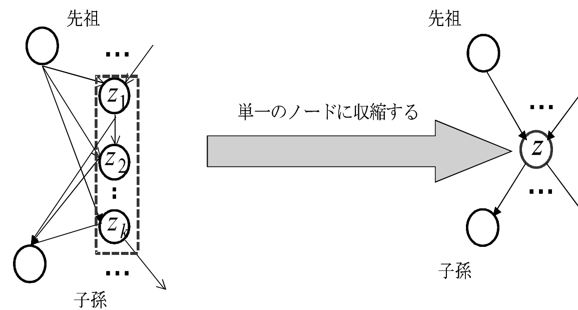


図 3.5: 基本ブロック化法の操作 O4

る．定義 3 で定義されたマルチプロセッサシステムの実行において，タスクグラフ G' はタスクグラフ G と等価である． □

証明: 補題 1 によって G' は非環式であるので G' はタスクグラフである．**操作** O**1-**O**4** の任意の操作を実行して，G と G' は実行前後のグラフとする．ノード z はノード z' の親であり，2 つのノードがブロックに結合することとする．G と G' を同じプロセッサで実行すると，G と G' の唯一の違いは G' 中の z と z' 間の通信時間 (z の書き出し時間と z' の読み込み時間) が 0 となることである．よって，G' と G は等価である． **Q.E.D.**

上記の操作を利用して，次のアルゴリズムを得る．このアルゴリズムは元のタスクグラフが小さいタスクグラフに変更することである．

≪ 基本ブロック化法を基づいてタスクグラフの変換 ≫

0° 入力: 元のタスクグラフ G; 出力: 変換されたグラフ G'.

1° $H \leftarrow G$.
2° H 中で MPC が存在すれば，H に**操作 O1** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．
3° H 中で PCP が存在すれば，H に**操作 O2** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．
4° H 中で PCS が存在すれば，H に**操作 O3** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．
5° H 中で PCPS が存在すれば，H に**操作 O4** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．
6° H 中で MPC または PCP または PCS または PCPS が存在すれば，2° へ．存在しなければ $G' \leftarrow H$ を行ってプログラムを停止する．

補題 2 と上記のアルゴリズムに基づいて次の定理が成り立つ．

[**定理 1**] G はタスクグラフ，G' は ≪ 基本ブロック化法を基づいてタスクグラフの変換 ≫ を利用して得られたタスクグラフとする．マルチプロセッサを利用して定義 3 で定義されたマルチプロセッサシステムの実行において，タスクグラフ G' はタスクグラフ G と等価である． □

証明: 補題 1 によって G' は非環式であるので，G' はタスクグラフである．H と H' をそれぞれ 2°-5° における**操作 O1-O4** の実行前後のグラフとする．ノード z はノード z' の親であり，この 2 つのノードをブロック B に結合したとすると，補題 2 より H と H' の唯一の違いは H' 中の z と z' 間の通信時間 (z の書き出し時間と z' の読み込み時間) が 0 となることである．よって，G' と G は等価である． **Q.E.D.**

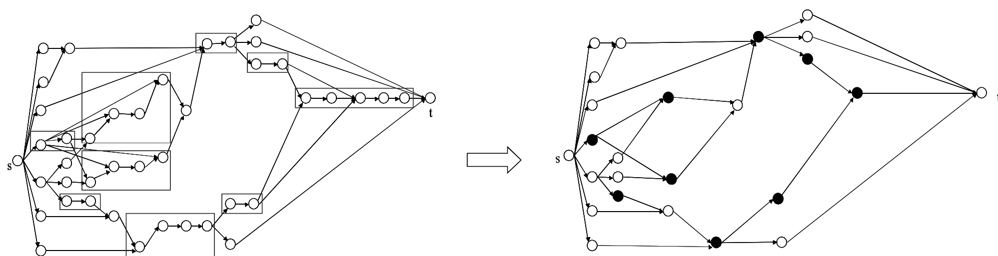


図 3.6: 基本ブロック化法の実行結果

図 3.6 は元のタスクグラフ (図の左のタスクグラフ) にアルゴリズム ≪ **基本ブロッ**

ク化法を基づいてタスクグラフの変換 ≫ を適用してタスクグラフのブロック化の結果を示している．なお，図中の矩形が縮約可能なブロックを表し，黒丸がブロックから縮約したノードを表す．各有向辺の通信時間を 1 (読み込み時間が 1，書き出し時間が 1) とする．アルゴリズムを実行する前のタスクグラフの総通信時間は 47, 実行した後のタスクグラフの総通信時間は 21 で，その差が 26 である．その総通信時間は 55%以上減少した．

3.2 追加ブロック化法の提案

この節では，通信時間がさらに短縮するために追加ブロック法を提案する．提案する追加ブロックは元のタスクグラフにアルゴリズム ≪ **基本ブロック化法を基づいてタスクグラフの変換** ≫ を適用して得られたタスクグラフに対してもう一度ブロックする方法である．そして，追加ブロック化法は最早最遅開始時刻の定義を利用することが必要である．

3.2.1 最早開始時刻と最遅開始時刻

以下の定義は，最早最遅開始時刻の概念である．常にスケジューリング問題中で使用される [44].

[**定義 7**] $G=(N,E)$ をタスクグラフとする．各ノード $z \in N$ の実行の期限時刻があり，z が z の実行の期限時刻前に実行を完了しなければならないのである．

- z の最早開始時刻 (Earliest Start Time) は z がそれより早くならない最も早く実行を開始できる時刻である．es_z で表示する．
- z の最遅開始時刻 (Latest Start Time) は z がそれより遅くならないように開始しなければならない時刻である．ls_z で表示する． □

図 3.7 が表示したノード z_x に対して次の補題は，z_x の最早と最遅開始時刻の計算を提供することができる．

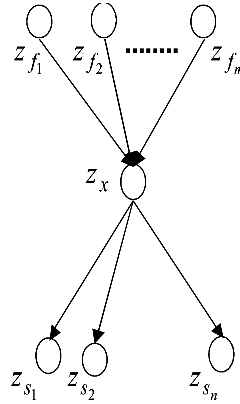


図 3.7: ノード z_x とその親子ノード

[**補題 3**] 図 3.7 の表示ように，Z_f=$\{z_{f_1}, z_{f_2}, \cdots, z_{f_m}\}$ をノード z_x の親の集合とし，Z_s=$\{z_{s_1}, z_{s_2}, \cdots, z_{s_n}\}$ をノード z_x の子の集合とする．

- z_x の最早開始時刻 (es_x) は次の式のように与えられる：

$$es_x = \max\{es_f + \sum_{z_g \in IP(z_f)} c_{g,f} + \gamma_f + \sum_{z_y \in IS(z_f)} c_{f,y} \mid z_f \in IP(z_x)\},$$

  ここでは，es_f は z_f の最早開始時刻であり，$\sum_{z_g \in IP(z_f)} c_{g,f}$ は z_f の読み込み時間であり，$\sum_{z_y \in IS(z_f)} c_{f,y}$ は z_f の書き出し時間である．

- z_x の最遅開始時刻 (ls_x) は z_x が開始しなければならない最も遅い時刻であり，次の式のように与えられる：

$$ls_x = \min\{ls_s \mid z_s \in IS(z_x)\} - (\sum_{z_s \in IS(z_x)} c_{x,s} + \gamma_x + \sum_{z_f \in IP(z_x)} c_{f,x}),$$

  ここでは，ls_s は z_s の最遅開始時刻であり，γ_x は z_x の実行時間であり，$\sum_{z_f \in IP(z_x)} c_{f,x}$ は z_x の読み込み時間であり，$\sum_{z_s \in IS(z_x)} c_{x,s}$ は z_x の書き出し時間である． □

証明: $z_f \in Z_f$ を z_x の任意の親とする．そして，es_f は z_f の最早開始時刻であり，$\sum_{z_g \in IP(z_f)} c_{g,f}$ は z_f のすべての入力からデータの読み込み時間 (通信時間) であり，γ_f

は z_f の実行時間である．また，$\sum_{z_y \in IS(z_f)} c_{f,y}$ は z_f のすべての出力へデータの書き出し時間 (通信時間) であり，$es_f + \sum_{z_g \in IP(z_f)} c_{g,f} + \gamma_f + \sum_{z_y \in IS(z_f)} c_{f,y}$ は z_f の最も早く実行を終了する時刻である．つまり z_f がこの時刻前に実行できないので，最も早い開始時刻は $es_x = \max\{es_f + \sum_{z_g \in IP(z_f)} c_{g,f} + \gamma_f + \sum_{z_y \in IS(z_f)} c_{f,y} \mid z_f \in IP(z_x)\}$ である．

P を最長パスとする．ノード z_f とノード z_x はこの最長パスに含まれる．そうすると，$es_f = ls_f$，$es_x = ls_x$．そして，既に証明された式 $es_x = es_f + \sum_{z_g \in IP(z_f)} c_{g,f} + \gamma_f + \sum_{z_y \in IS(z_f)} c_{f,y}$ を利用して次の最早最遅開始時刻が得られる．

$$ls_x = ls_f + \sum_{z_g \in IP(z_f)} c_{g,f} + \gamma_f + \sum_{z_y \in IS(z_f)} c_{f,y},$$

$$ls_f = ls_x - \sum_{z_g \in IP(z_f)} c_{g,f} - \gamma_f - \sum_{z_y \in IS(z_f)} c_{f,y}.$$

Q.E.D.

上記の補題を利用して，各ノードの最早開始時刻が得られる．ノード z の最早開始時刻はノード s からノード z までの最長パスの長さである．最遅開始時刻について，まず最長パスを探し，それから，この最長パスに含まれる各ノード z_x に対して ls_x=es_x にして，その後，上記の補題を利用して最長パスに含まれるノードを除いて，すべての他のノードの最遅開始時刻を得る．

3.2.2 追加ブロック化法

図 3.8 は 4 種類の親子ノードの接続パターンを示している：(1) 多対一親子接続パターン，(2) 一対多親子接続パターン，(3) ある親ノードが別の子ノードを持つ多対多親子接続パターン，(4) ある子ノードが別の親ノードを持つ多対多親子接続パターン．アルゴリズム ≪ **基本ブロック化法を基づいてタスクグラフの変換** ≫ を適用して変更したタスクグラフはこの 4 種類の親子の接続パターンで構成する．

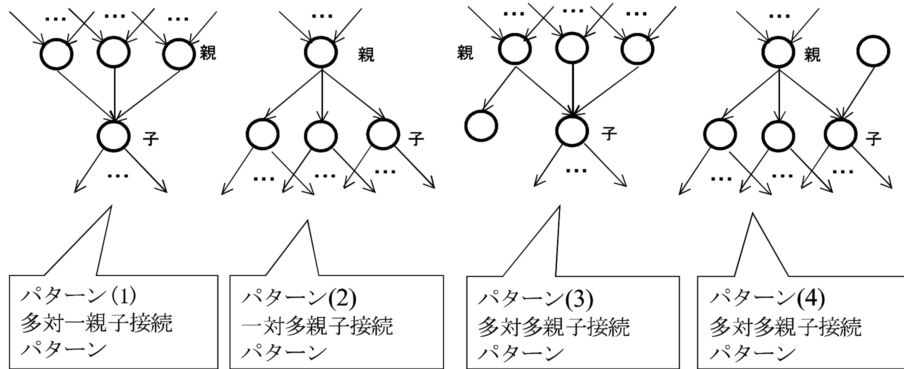


図 3.8: 親子関係の 4 つのパターン

追加ブロック化法の基本的な考え方として図 3.9 のように，パターン (1) と (3) に対して，子ノードが複数の親ノードを存在する場合には複数の親ノードの中で実行開始時刻が最も遅い親ノードを選択し，選択した親ノードとこの子ノードをブロックにする；パターン (1) と (3) に対して，親ノードに複数の子ノードが存在する場合には複数の子ノードの中で実行開始時刻が最も早い子ノードを選択し，選択した子ノードとこの親ノードをブロックにする．

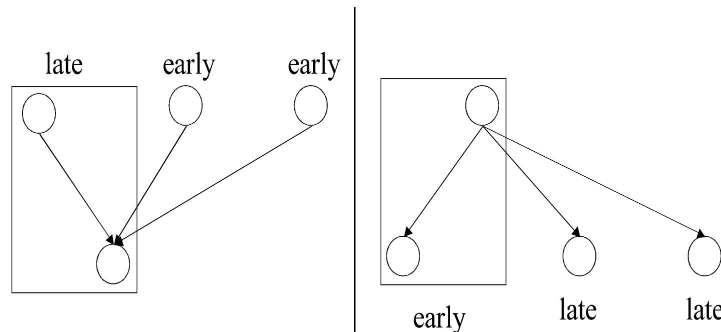


図 3.9: 追加ブロック化法にブロックできる場合

次は，図 3.8 が示した 4 つのパターンにそれぞれ対応する 4 つ追加操作 (**操作 AO1-AO4**) を紹介する．

[**操作 AO1**] 図 3.10 のように $z_1, z_2, \cdots, z_n$ をノード z の親ノードとする．(i)(z_1, z) を除いて z_1 から z への他のパスが存在しない；(ii)z_1 の最早開始時刻が $es_{(z_1)} > es_{(z_j)}$

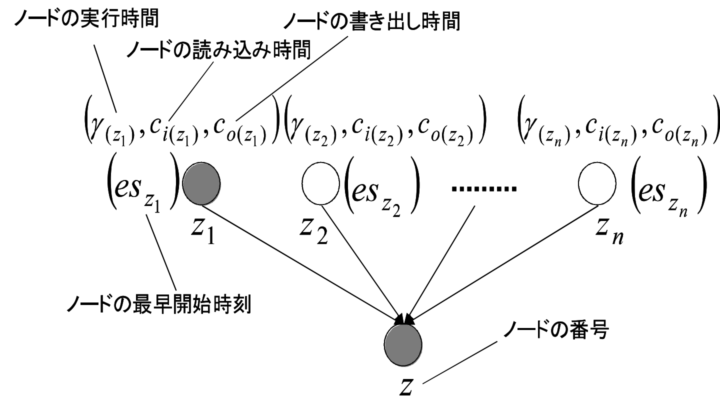


図 3.10: 追加ブロック化法のパターン (1)

$(j = 2, 3 \cdots n)$ を満たす ; (iii)z_1 の最遅開始時刻が $ls_{(z_1)} \geq es_{(z_j)} + c_{i(z_j)} + \gamma_{(z_j)} + c_{o(z_j)}$ $(j = 2, 3 \cdots n)$ を満たす．上記の 3 つ条件を満たしたら，z_1 と z がブロック B に構成して B を縮約して単一のノードに変更する． □

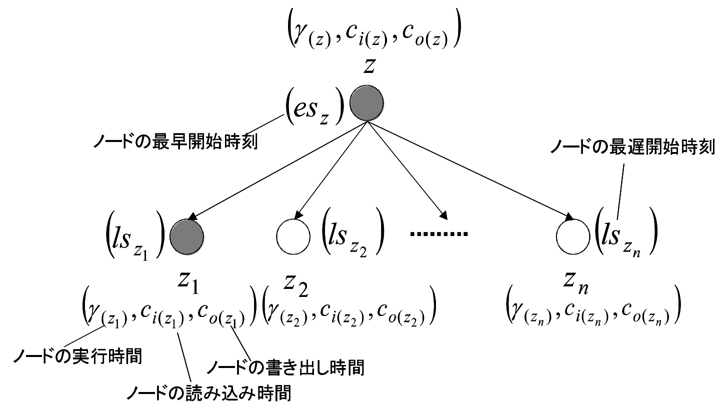


図 3.11: 追加ブロック化法のパターン (2)

[操作 AO2] 図 3.11 のように $z_1, z_2, \cdots, z_n$ をノード z の子ノードとする．(i)(z, z_1) を除いて z から z_1 への他のパスが存在しない ; (ii)z_1 の最遅開始時刻が $ls_{(z_1)} < ls_{(z_j)}$ $(j = 2, 3 \cdots n)$ を満たす ; (iii)z_j $(j = 2, 3 \cdots n)$ の最遅開始時刻が $ls_{(z_j)} \geq es_{(z)} + c_{i(z)} + \gamma_{(z)} + c_{o(z)} + \gamma_{(z_1)} + c_{o(z_1)} - c_{i(z_1)}$ を満たす．上記の 3 つ条件を満たしたら，z と z_1 がブロック B に構成して B を縮約して単一のノードに変更する． □

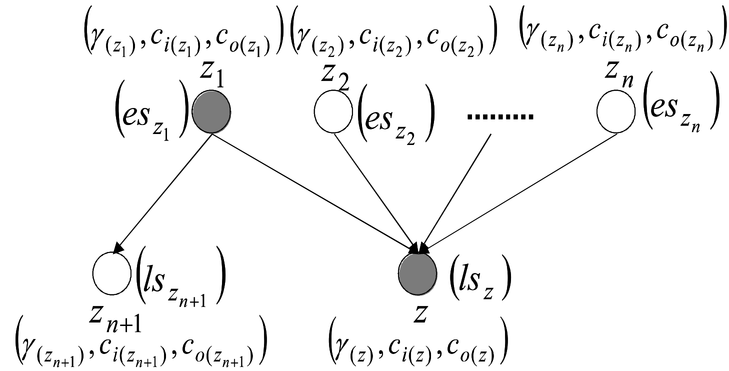


図 3.12: 追加ブロック化法のパターン (3)

[**操作 AO3**] 図 3.12 のように $z_1, z_2, \cdots, z_n$ をノード z の親ノードとする．(i)(z_1, z) を除いて z_1 から z への他のパスが存在しない；(ii)z_1 の最早開始時刻が $es_{(z_1)} > es_{(z_j)}$ $(j = 2, 3 \cdots n)$ を満たす；(iii)z_1 の最遅開始時刻が $ls_{(z_1)} \geq es_{(z_j)} + c_{i(z_j)} + \gamma_{(z_j)} + c_{o(z_j)}$ $(j = 2, 3 \cdots n)$ を満たす；(iv)z を除いて z_1 の任意の子ノード z_{n+1} の最遅開始時刻が $ls_{(z_{n+1})} \geq es_{(z_1)} + c_{i(z_1)} + \gamma_{(z_1)} + c_{o(z_1)} + c_{i(z)} + \gamma_{(z)} + c_{o(z)} - 2c_{(z_1,z)}$ を満たす．上記の 4 つ条件を満たしたら，z_1 と z がブロック B に構成して B を縮約して単一のノードに変更する． □

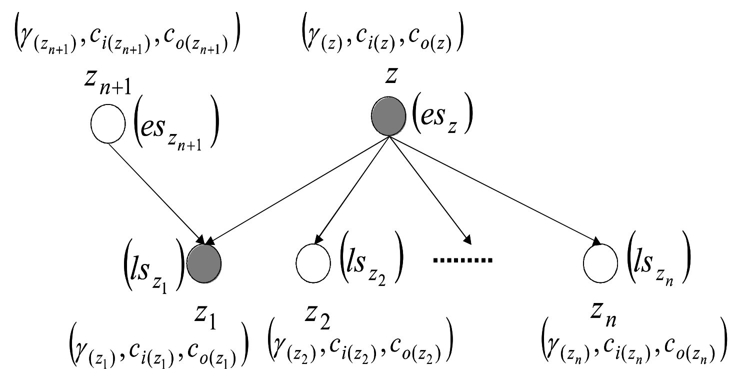


図 3.13: 追加ブロック化法のパターン (4)

[**操作 AO4**] 図 3.13 のように $z_1, z_2, \cdots, z_n$ をノード z の子ノードとする．(i)(z, z_1) を除いて z から z_1 への他のパスが存在しない；(ii)z_1 の最遅開始時刻が $ls_{(z_1)} < ls_{(z_j)}$

$(j = 2, 3 \cdots n)$ を満たす；(iii)z_j $(j = 2, 3 \cdots n)$ の最遅開始時刻が $ls_{(z_j)} \geq ex_{(z)} + c_{i(z)} + \gamma_{(z)} + c_{o(z)} + c_{i(z_1)} + \gamma_{(z_1)} + c_{o(z_1)} - 2c_{(z,z_1)}$ を満たす；(iv)z を除いて z_1 の任意の親ノード z_{n+1} に対して z の最遅開始時刻が $ls_{(z)} \geq es_{(z_{n+1})} + c_{i(z_{n+1})} + \gamma_{(z_{n+1})} + c_{o(z_{n+1})}$ を満たす．上記の 4 つ条件を満たしたら，z と z_1 がブロック B に構成して B を縮約して単一のノードに変更する． □

次の補題は**操作 AO1-AO4** について明らかに満たされている．

[**補題 4**] G はタスクグラフとする．G に対して**操作 AO1-AO4** の任意の操作を適用して得られたブロック B は縮約可能なブロックである． □

証明: まず，**操作 AO2** と**操作 AO4** の場合について証明する．ノード z は z' の親であり，2 つのノードが同じブロック B に結合することとする．ブロック B を単一のノードに変更してタスクグラフに有向閉路ができた場合は有向枝 (z, z') 以外に z から z' までのパスが存在することになる．このようなパスが存在した場合，上記の**操作 2** と**操作 4** の条件 (i) によりノード z と z' は同じブロックにすることができないことになる．同様に，操作 **AO1** と**操作 AO3** の場合も，このようなパスが存在した場合，ノード z' と z は同じブロックにすることができない．よって，ブロック B は単一のノードに変更した時に，タスクグラフに有向閉路が存在しない．したがって，この補題が成り立つ． **Q.E.D.**

≪ 追加ブロック化法を基づいてタスクグラフの変換 ≫

0° 入力: タスクグラフ G; 出力: 変換されたグラフ G'.

1° $H \leftarrow G$.

2° H 中でパターン (1) が存在すれば，H に**操作 AO1** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．

3° H 中でパターン (2) が存在すれば，H に**操作 AO2** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．

4° H 中でパターン (3) が存在すれば，H に**操作 AO3** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．

5° H 中でパターン (4) が存在すれば，H に**操作 AO4** を実行して変更されたグラフ H' を得る．$H \leftarrow H'$ を行う．

6° H 中でパターン (1) またはパターン (2) またはパターン (3) またはパターン (4) が存在すれば，2° へ．存在しなければ $G' \leftarrow H$ を行ってプログラムを停止する．

補題 4 と上記のアルゴリズムに基づいて次の定理が成り立つ．

[**定理 2**] G はタスクグラフ，G' は ≪ 追加ブロック化法を基づいてタスクグラフの変換 ≫ を利用して得られたタスクグラフとする．マルチプロセッサを利用して定義 3 で定義されたマルチプロセッサシステムの実行において，タスクグラフ G' はタスクグラフ G と等価である． □

証明: 補題 4 によって G' は非環式であるので，G' はタスクグラフである．H と H' をそれぞれ 2°-5° における**操作 AO1-AO4** の任意の操作で実行した時の実行前後のグラフとする．ノード z とノード z' は親子関係であり，2 つのノードがブロック B に結合したとする．H と H' を同じプロセッサで実行すると，H と H' の唯一の違いは H' 中の z と z' 間の通信時間が 0 となることである．よって，G' と G は等価である． **Q.E.D.**

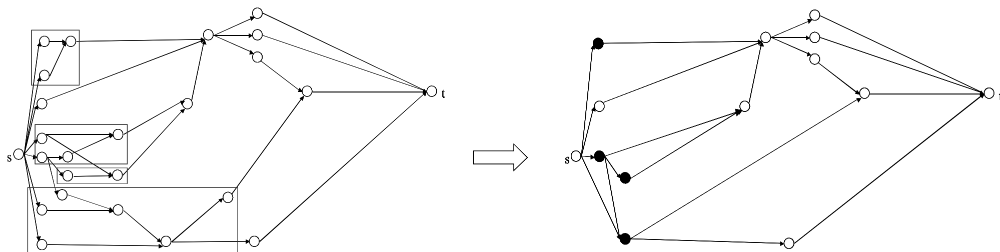


図 3.14: 追加ブロック化法の実行結果

図 3.14 の左のタスクグラフは元のタスクグラフにアルゴリズム ≪ **基本ブロック化法を基づいてタスクグラフの変換** ≫ を適用して得られたタスクグラフ (図 3.6 右のタスクグラフ) である．図 3.14 の右のタスクグラフは上記のアルゴリズム ≪ **追加ブロック化法を基づいてタスクグラフの変換** ≫ を利用して矩形で表示したブロックが単一のノードに変更してさらに変更したタスクグラフである．結果としてタスクグラフの通信時間は 47 から 18 に変更して，61%以上減少した．よって，提案したブロック化法は例のタスクグラフに対して高い有効性を示している．

第 4 章

再ブロック化法の提案

本章では再ブロック化法を提案する．まず最早最遅開始時刻によってタスクグラフのスケジュール結果を求める LST-EST 法を説明し, 次にそのスケジュール結果に基づいて再ブロック化法を提案する.

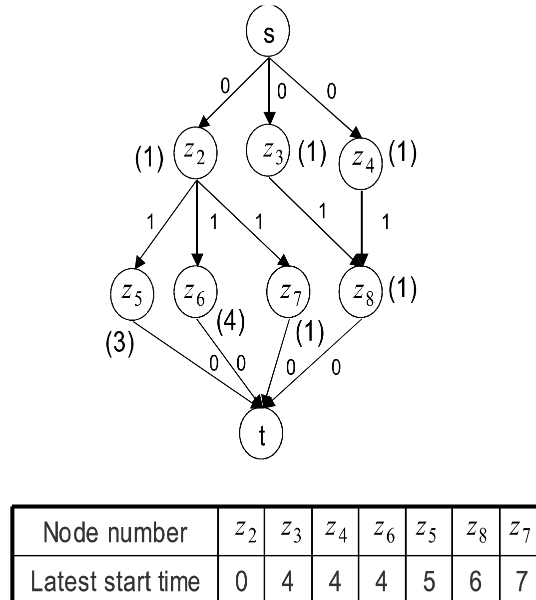
4.1 最早最遅開始時刻に基づいたスケジューリング法

ここで提案するスケジューリング法は再ブロック化法に用いられる．現在，様々なスケジューリング法が存在している [45, 46, 47]. 本論文では最早開始時刻と最遅開始時刻に基づいたスケジューリング法 (LST-EST 法) を提案する．LST-EST 法はスケジュールの長さを増加させずに，ある条件に基づいてスケジューリング中の隣接する 2 つのノードを親子関係であるノードに変更することができる.

最早開始時刻と最遅開始時刻の定義は既に 3 章の 3.2.1 節に定義された．ここでは定義された最早最遅開始時刻を利用してスケジューリング法を提案する．まず，最遅開始時刻の計算式を利用して各ノードの最遅開始時刻の値を求める．次に得られた最遅開始時刻の値を昇順に並べてタスクグラフの優先リストを生成する．優先リストに含まれるすべてのノードに対して，最遅開始時刻の値が小さければ小さいほどその優先度が高い．図 4.1 は優先リストの例を示している．このタスクグラフの優先リストは $L{=}z_2, z_3, z_4, z_6, z_5, z_8, z_7$ である.

次は LST-EST 法を紹介する．この方法はタスクグラフのスケジュールの長さを増加させずに，親子関係であるノードが結合した構造であるブロックの数をできるだけ多くするために使われる.

このスケジューリング法の基本の考えは，ノードとノード間の通信時間を省くこ



Node number	z_2	z_3	z_4	z_6	z_5	z_8	z_7
Latest start time	0	4	4	4	5	6	7

図 4.1: タスクグラフの優先リスト

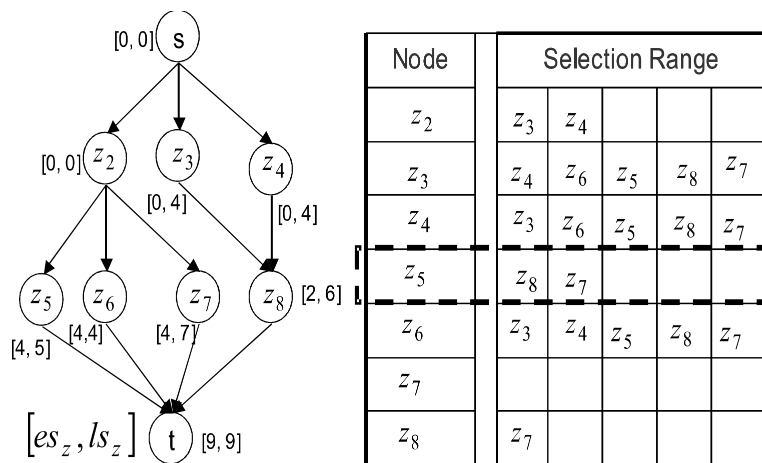
とができるように，親子関係のノードを (隣接的なノード) できるだけ同じプロセッサで割り当てる．それには優先リストの実行順を変更する必要があるが，勝手に実行順位を変更すると，スケジュールの長さが長くなる．この問題を解決するために，ノードの選択範囲 (Selection Range) を使用する必要がある．

ノードの選択範囲とは，ノードを選択することによって決定される範囲である．タスクグラフのスケジュール結果を増加しないように，ノード z_i の選択範囲中に含まれるノードとノード z_i の実行順位を変えることができる．

ノードの選択範囲を使うルールは次のように説明する．優先順位に基づいてノード z_i をノード z_j と同じプロセッサで実行することによって，ノード z_i と z_j が親子関係の場合はノード z_i の選択範囲を使わずに，ノード z_i と z_j が親子関係ではない場合はノード z_i の選択範囲を使う．ノード z_i の選択範囲に含まれるすべての実行可能なノードを調べて，ノード z_j と親子関係であるノードが存在すれば，ノード z_i と変更して優先実行する．

ノード z_i とノード z_j が次の条件を満たしたら，この 2 つのノードが同じ選択範囲に含まれる :” ノード z_i の最遅開始時刻がノード z_j の最早最遅開始時刻の間に含ま

れる．$es_{z_j} \leq ls_{z_i} \leq ls_{z_j}$ ような式で表示する ”.
図 4.2 はタスクグラフのすべてのノードの選択範囲を示している．
優先リスト $L = z_2, z_3, z_4, z_6, z_5, z_8, z_7$ に表示されたように選択範囲に含まれるノードが順に表示される．ノードの選択範囲の具体的な使い方は次の例で説明する．
選択範囲の使用法については後述する．



Node	Selection Range				
z_2	z_3	z_4			
z_3	z_4	z_6	z_5	z_8	z_7
z_4	z_3	z_6	z_5	z_8	z_7
z_5	z_8	z_7			
z_6	z_3	z_4	z_5	z_8	z_7
z_7					
z_8	z_7				

図 4.2: 各ノードの選択範囲

ここに提案した LST-EST 法はノード選択範囲を利用する．ノード z は実行を終了したことにする．(1) まず優先リスト L から優先順位によって次の実行可能ノードの z' を選択する；(2)z の子は z' であることをチェックする；(3)z の子は z' である場合，z' を選び，z と同じブロックに割り当てる；(4)z の子は z' ではない場合，(i)z' の選択範囲から選択範囲の順に z の子があるかどうかをチェックする；(ii)z の子が存在する場合はそれを選ぶ；(iii)z の子が存在しない場合は z' を選ぶ.

図 4.3 は LST-EST 法の実行結果を示している．ここでは，z_2 はプロセッサ p_1 で実行し終わって，z_6 は優先リスト L によって選択される．z_2 の子はちょうど z_6 であるから z_6 は p_1 に割り当てられる．一方で z_4 がプロセッサ p_2 で実行し終わる時に優先リスト L によって z_5 を選択するはずであるが，z_5 は z_4 の子ではないので，z_5 の選択範囲の中で z_8 が z_4 の子である．よって，z_8 を選ぶ．得られたスケジュール結果によって，(z_2, z_6) と (z_4, z_8) はそれぞれ親子関係であり，個々同じプロセッサで実行される．

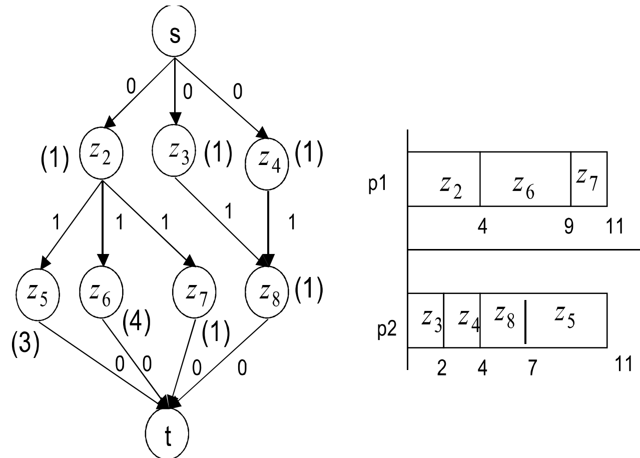


図 4.3: ノードの選択範囲を使う LST-EST 法

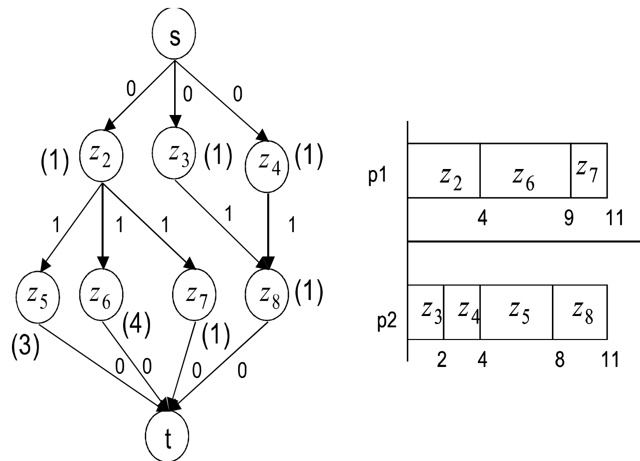


図 4.4: ノードの選択範囲を使わずに普通のスケジューリング法

図 4.1 のタスクグラフの例によって，図 4.4 と 4.3 のスケジュールの長さは同じで 11 である．しかし，LST-EST 法を利用して得られたスケジュール結果は (z_2, z_6) と (z_4, z_8) 二つの親子関係のノードペアーを発見し，その結果は別の方法 (図 4.4) よりスケジュールの長さが長くならずにもっと多くの親子関係のノードペアー構造ができる．よって，LST-EST 法で得られたスケジュール結果に対して，次に提案する再ブロック化法にもっとも適用できる．

4.2 再ブロック化法

本節では，マルチプロセッサに基づいてスケジュール結果を短縮するために，ブロック化法 (図 4.5) で実行されたタスクグラフにもう一度ブロック化する再ブロック化法を提案することである．

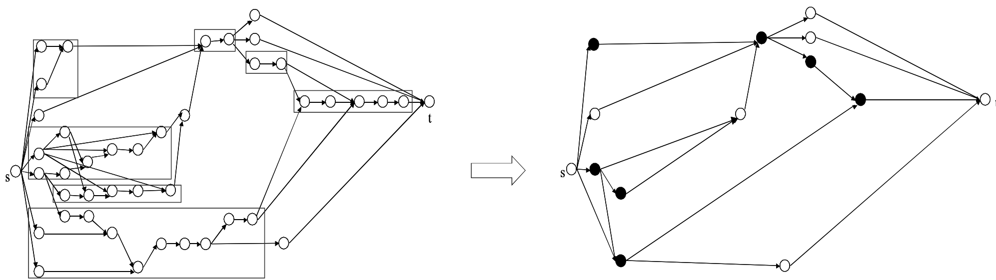


図 4.5: 元のタスクグラフにブロック化法の実行結果

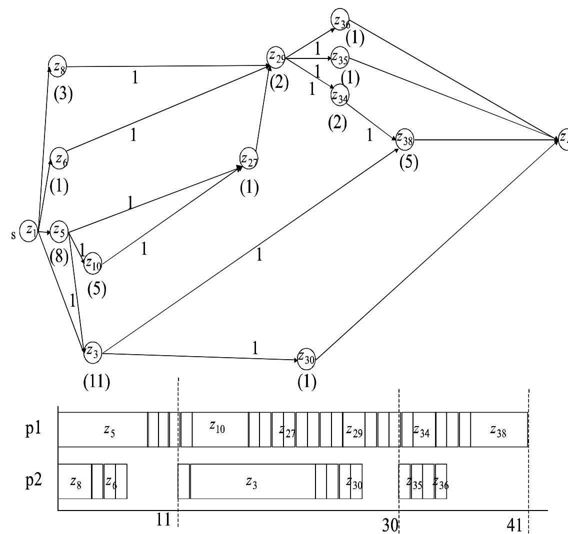


図 4.6: 実行されたタスクグラフのスケジュール結果

次には再ブロック化法の具体的な手順を紹介する．

[**操作 1**] ブロック化法で実行されたタスクグラフに LST-EST 法を利用してスケジュール結果を求める (図 4.7)．

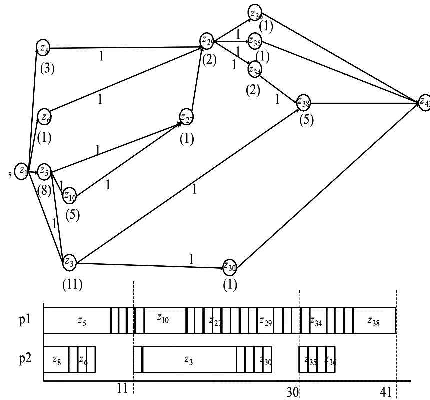


図 4.7: 実行されたタスクグラフのスケジュール結果

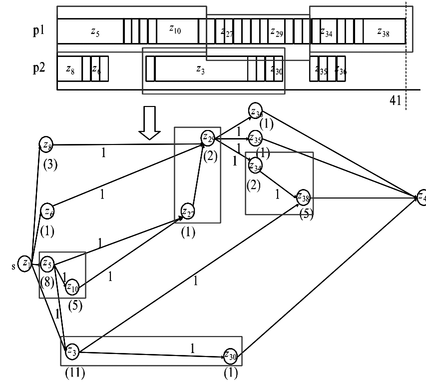


図 4.8: ノード z と z' がブロック B に構成

[**操作 2**] 得られたスケジュール結果に基づいて，同じプロセッサで実行されるノードに対してまとめてノードリストを作る．図 4.7 のようにプロセッサ p_1 のノードリストは $z_5, z_{10}, z_{27}, z_{29}, z_{34}, z_{38}$ である．プロセッサ p_2 のノードリストは $z_8, z_6, z_3, z_{30}, z_{35}, z_{36}$ である．

[**操作 3**]　各ノードリストに対して，第一のノード z 及びノードリスト中の z の隣接ノード z' から探索し始まる．ノード z と z' が次の条件を満たしたら，z と z' がブロック B に構成して B を縮約して単一のノードに変更する (図 4.8, 4.9)．(i)z と z' は親子関係ノード ; (ii) 得られたスケジュール結果に基づいてノード z の実行開始時刻 $st_{(z)}$ は $st_{(z)} \geq et_{(IP(z'))}$ を満たす．ここでは $et_{(IP(z'))}$ がノード z' のすべての親ノードの実行終了時刻である ; (iii) 得られたスケジュール結果に基づいてさらにノード z' の実行終了時刻 $et_{(z')}$ は $et_{(z')} - 2c_{z,z'} \leq st_{(IS(z))}$ を満たす．ここでは $st_{(IS(z))}$ がノード z のすべての子ノードの実行開始時刻であり，$c_{z,z'}$ が z と z' 間の通信時間である．

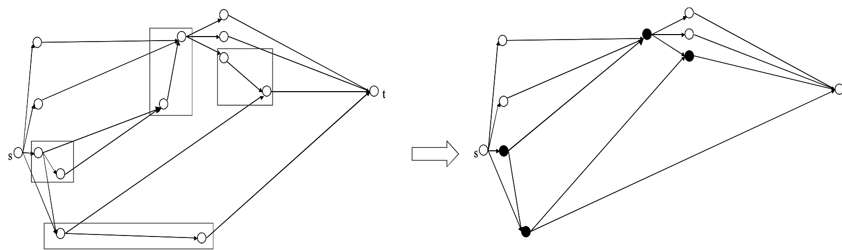


図 4.9: ブロック B は単一のノードに縮約

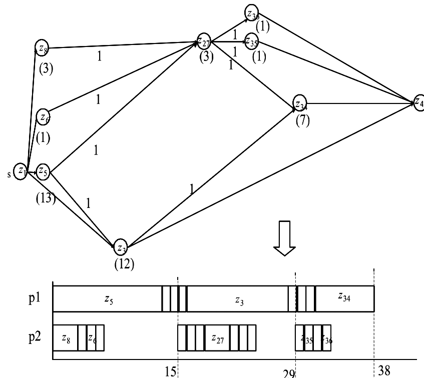


図 4.10: LST-EST 法に基づいてタスクグラフ G' のスケジュール結果

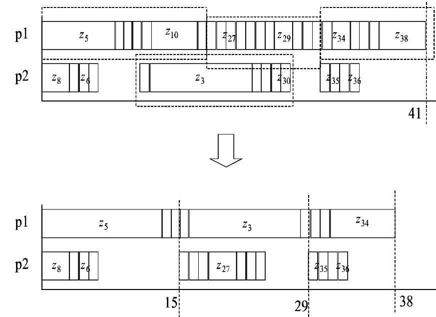


図 4.11: 再ブロック化法の結果

操作 3 はタスクグラフの非循環性を保証する．したがって，次の補題が成り立つ．

[**補題 5**] G をタスクグラフとする．G を**操作 3** で実行して得られたブロック B は縮約可能なブロックである． □

証明: ノード z は z' の親であり，2 つのノードが同じブロック B に結合したとする．ブロック B を単一のノードに変更した時に，タスクグラフに有向閉路ができたとすると，有向枝 (z, z') 以外に，z から z' までのパスが存在することになる (図 4.12)．実際には，ノード z_k はノード z' の親であり，ノード z_1 はノード z の子である．このようなパスが存在した場合，上記の**操作 3** の条件 (ii) と (iii) により，ノード z と z' は同じブロックに含むことができない．よって，ブロック B は単一のノードに変更した時に有向閉路が存在しない．したがって，この補題が成り立つ． **Q.E.D.**

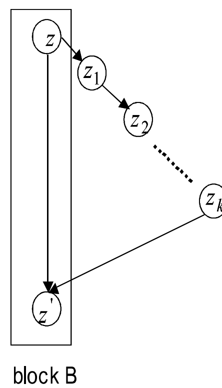


図 4.12: 親子関係のノードと別のパス

≪ 再ブロック化法を基づいてタスクグラフの変換 ≫

0° 入力: タスクグラフ G; 出力: 変換されたグラフ G'.

1° $H \leftarrow G$.

2° H に**操作 1-3** で実行して，グラフ H' を得る．

3° グラフ H と同じであるグラフ H' が存在したら $G' \leftarrow H'$ を行ってプログラムを停止する；或いは $H \leftarrow H'$ を行って，2° へ.

補題 5 と上記のアルゴリズムより，次の定理が成り立つ.

[**定理 3**] G はタスクグラフ，G' は ≪ 再ブロック化法を基づいてタスクグラフの変換 ≫ を適用して得られたタスクグラフとする．マルチプロセッサを利用して定義 3 で定義されたマルチプロセッサシステムの実行において，タスクグラフ G' はタスクグラフ G と等価である. □

証明: 補題 5 によって G' は非環式であるので，G' はタスクグラフである．H と H' を 2° における**操作 3** の 1 回実行した時の実行前後のグラフとする．ノード z はノード z' の親であり，2 つのノードがブロック B に結合したとする．H と H' を同じプロセッサで実行した時に，H と H' の唯一の違いは H' 中の z と z' 間の通信時間 (z の書き出し時間と z' の読み込み時間) が 0 となることである．よって，G' と G は等価である. **Q.E.D.**

図 4.10 は LST-EST 法に基づいて得られたスケジュール結果を利用した再ブロック化法の結果である．結果として，LST-EST 法を利用した再ブロック化法によってタスクグラフのスケジュール結果は 41 から 38 に減少した (図 4.11)．よって，提案した再ブロック化法はタスクグラフの実行時間をさらに短縮することができる.

第 5 章

シミュレーション結果と考察

本章で, 提案したブロック化法と再ブロック化法を評価するためのシミュレーション結果について述べる．まず，Matlab を用いた Simulink モデルの隣接情報を抽出し，タスクグラフの各情報を求める．次は得られたタスクグラフを利用し，提案した方法でシミュレーション実験を行い，提案したブロック化法と再ブロック化法の有効性を確認する．最後に，シミュレーション用の各実行用プログラムを紹介し，実行結果を示す.

5.1 Matlab を用いた Simulink モデルの隣接情報抽出

付録 I は MATLAB/Simulink でモデル化される車両制御システムである．結果と考察についてこの車両制御システムモデルのタスクグラフを利用して実験を行う.

本論文が定義したタスクグラフは有向閉路がないグラフである (DAG)．実験用の車両制御システムモデルはサブシステムが存在し，閉路線または多重入力線が存在することが分かる．このシステムはタスクグラフに変更する時に，車両制御システムモデル上の全てのサブシステムモデルを展開するだけでなく，ループ入力線または多重入力線を変更することも必要である.

ここでは，まずタスクグラフを表現するタスクグラフの行列，次に閉路と多重辺の処理，そして最後にサブシステムモデルの展開について紹介する.

5.1.1 タスクグラフの行列について

タスクグラフの行列には，すべてのノード数，各ノードの番号や実行時間，通信時間，他ノードとの隣接関係が必要である．タスクグラフの行列の形式を以下の通りに定めることにした．**ちなみに，MATLAB/Simulink 中の中身はブロックと呼び，提案したブロック化法または再ブロック化法中のブロックとは違う**．

- 一行目はソースノードとシンクノードを除いたノードの総数とする．
- ソースノード番号は一番最初のノード番号，シンクノード番号は一番最後のノード番号とする．
- ソースノード，シンクノードの実行時間は 0(単位時間)，それ以外の実行時間は 1(単位時間) とする．
- ソースノードから各ノードへの通信時間及び，各ノードからシンクノードへの通信時間は 0 (単位時間)，それ以外のノード間の通信時間は 1(単位時間) とする．
- サブシステムブロックの実行時間及び，通信時間，親ノードの数はそれぞれ 0(単位時間) とする．
- サブシステム内の Inport ブロック及び Outport ブロックは通常のブロック入力，出力ポートであるので，実行時間及び，通信時間 0 である．

図 5.1 は図 2.1 のタスクグラフの行列を示している．ちなみに，このタスクグラフの実行時間と通信時間はランダムの数字で設定した．また，タスクグラフの行列に基づいて各ノードの情報を用いてタスクグラフを表現できる．

5.1.2 閉路と多重辺の処理について

タスクグラフの定義によれば，それは閉路や多重辺を有してはならない．しかし，実際の Simulink モデルには閉路や多重辺が存在する．ゆえに，タスクグラフの行列上より，閉路または多重辺となりうる信号線の情報を削除しなければならない．

タスクグラフ中のノードの数	8								
ソースノードs	0	0	0						
ノード z_1	1	5	1	0	0				
ノード z_2	2	8	1	0	0				
ノード z_3	3	9	2	0	0	1	1		
ノード z_4	4	10	1	1	1				
ノード z_5	5	8	1	2	1				
ノード z_6	6	8	1	3	2				
ノード z_7	7	8	1	4	1				
ノード z_8	8	9	2	5	1	7	2		
シンクノードt	9	0	3	6	0	7	0	8	0

各ノードの番号 / 各ノードの実行時間 / 親の数 / 親の番号 / 親からの通信時間

図 5.1: 図 2.1 のタスクグラフの行列の形式

閉路の処理について

タスクグラフの行列より，閉路となりうる信号線の情報を削除するためには，タスクグラフの行列の中から閉路となりうる信号線の情報を見つけ出さなければならない (図 5.2)．そこで，本論文では，閉路の探索手段として，深さ優先探索法 [48, 49] を用いることにした．深さ優先探索法とは，始点 (ソースノード) からの距離には関係なく，訪問できる辺がある限りできるだけ「深く」訪問するという方法である．
この探索法に基づいて以下の手順で閉路の処理を行う．

[手順 1]

ソースノードから順にブロックを検索していく，一度検索したブロックは「訪問済み」のチェックを入れる．

[手順 2]

探索中のブロックに子ブロックが存在しない場合,「探索済み」のチェックを入れる．また，子ブロックが存在する場合でもその子ブロックに「探索済み」のチェック

がついている場合は，子ブロックは存在しないものと見なす．

[手順 3]

探索中のブロックが「訪問済み」のチェックが入っているブロックが，子ブロックである場合，その子ブロックの親の情報からそのブロックの情報を削除する．

[手順 4]

ソースノードに「探索済み」のチェックが入ったら探索終了とする．

また，子ブロックの親の情報からそのブロックの情報を削除する際に以下の 3 つ規定がある．

規定 1 : 対象ブロックの子ブロックの親ブロックの個数を 1 を減らす．

規定 2 : 対象ブロックの子ブロックの親ブロックを削除した後で，タスクグラフの行列を作成する際，子ブロックを持たないブロック，即ち出力ポートが存在しないブロックはすべてシンクノードに繋げている．ゆえに，閉路の処理をする過程で子ブロックの個数が 0 となったブロックもシンクノードに繋げなければならない．

規定 3 : 対象ブロックの子ブロックの個数が 0 になった場合シンクノードに繋げる．

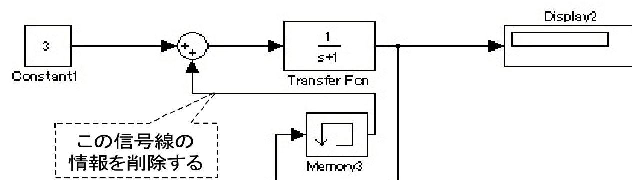


図 5.2: 閉路の処理

多重辺の処理について

多重辺の処理は，親ブロックを複数持つブロックに対して，親が重複しているかを確かめ，重複する親の番号を 1 つにする (図 5.3)．その際の規定は以下の 2 つである．

規定 1 : 対象ブロックの子ブロックの親ブロックの個数は重複の回数によって減らす.
規定 2 : 対象ブロックの子ブロックの親ブロックの情報を削除した後で，タスクグラフの行列を作成する際，重複の親ブロックの情報をただ 1 回入力する.

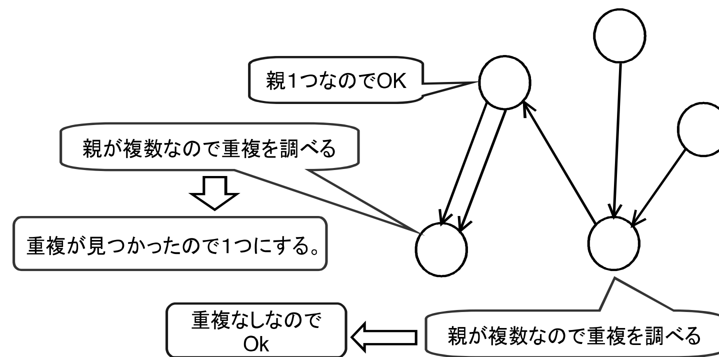


図 5.3: 多重辺の処理

5.1.3 サブシステムブロックの展開について処理

サブシステムブロックの展開は Simulink モデル上にある全てのサブシステムブロックを展開し，その状態の Simulink モデルの隣接情報を抽出し，タスクグラフの行列を作成する処理である．サブシステムブロックの展開の処理の流れは以下の通りである．

[**処理 1**] 各ブロックの親情報を取得.

Simulink モデル上にある全てのブロックに番号を振り当てる．各ブロック間の接続情報を調べ，その親ブロックの番号を保存する．この保存した情報をブロックの親情報と呼ぶ．以下の手順で親情報を取得していく．

手順 1 :

サブシステムブロックの入力ポート番号を取得する．取得した入力ポート番号に対応するサブシステムブロック内の Inport ブロックを探索する．

手順 2 :

信号線の情報を元に親ブロックを探索し，出力ポート番号を取得する．

手順 3 :

取得した出力ポート番号に対応するサブシステムブロック内の Outport ブロックを探索する．

手順 4 :

(1) 対象のブロックがサブシステムブロックで, 親ブロックもサブシステムブロックの場合は，手順 1 で求めた Inport ブロックの親情報に，手順 3 で探索したサブシステムブロック内の Outport ブロック番号と出力ポート番号を追加する；

(2) 対象のブロックがサブシステムブロックで, 親ブロックがサブシステムブロックでない場合は，手順 1 で求めた Inport ブロックの親情報に，手順 2 で探索した親ブロックのブロック番号と出力ポート番号を追加する；

(3) 対象のブロックがサブシステムブロックではなく，親ブロックがサブシステムブロックの場合は，対象のブロックの親情報に，手順 3 で探索したサブシステムブロック内の Outport ブロック番号と出力ポート番号を追加する；

(4) 対象のブロックと親ブロックがサブシステムブロックでない場合は，対象のブロックの親情報に，手順 2 で探索した親ブロックのブロック番号と出力ポート番号を追加する．

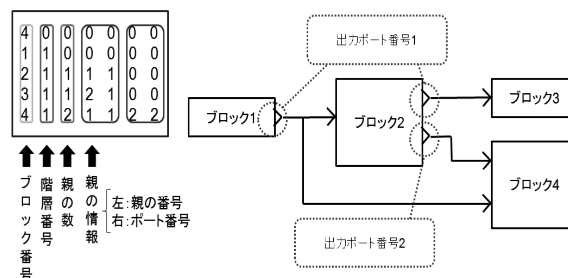


図 5.4: 各ブロックの親情報を取得する処理

[処理 2] Inport ブロックと Outport ブロックの処理.

処理 2.1：

処理対象の Inport ブロックの親ブロックの番号をその Inport ブロックの子ブロックの親情報に追加する．そのとき，Inport ブロックの子ブロックの親情報から Inport ブロックの番号を削除する．

処理 2.2：

処理対象の Outport ブロックの親ブロックの番号をその Outport ブロックの子ブロックの親情報に追加する．そのとき，Outport ブロックの子ブロックの親情報から Outport ブロックの番号を削除する．

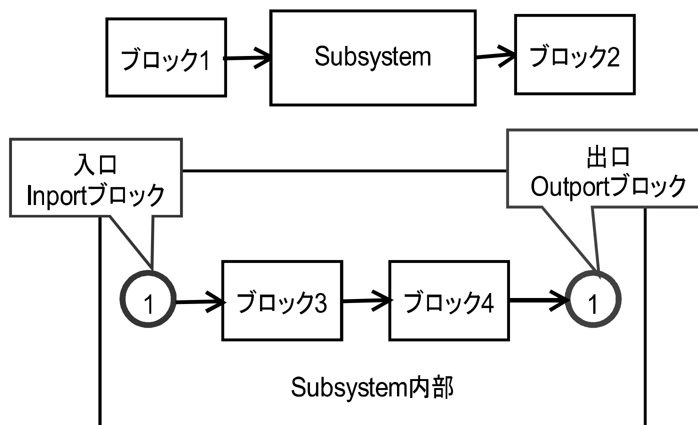


図 5.5: サブシステム内の Inport と Outport

[処理 3]　サブシステムブロックの展開．

ブロックの親情報によってサブシステムブロックを展開していく．展開する順番は，サブシステムを持たない階層から行く．処理の手順は以下の通りである．

手順 1：展開するサブシステムブロックに接続のある信号線を削除する.

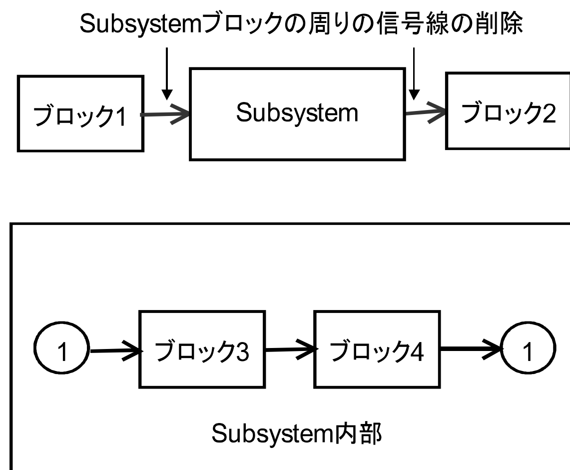


図 5.6: サブシステムブロックの展開の手順 1

手順 2：展開領域を確保する.

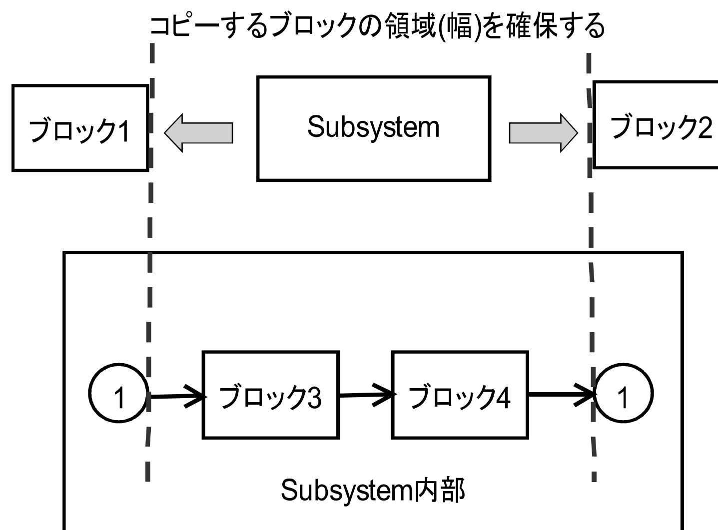


図 5.7: サブシステムブロックの展開の手順 2

手順 3：ブロックを貼り付け (コピー) して移動する.

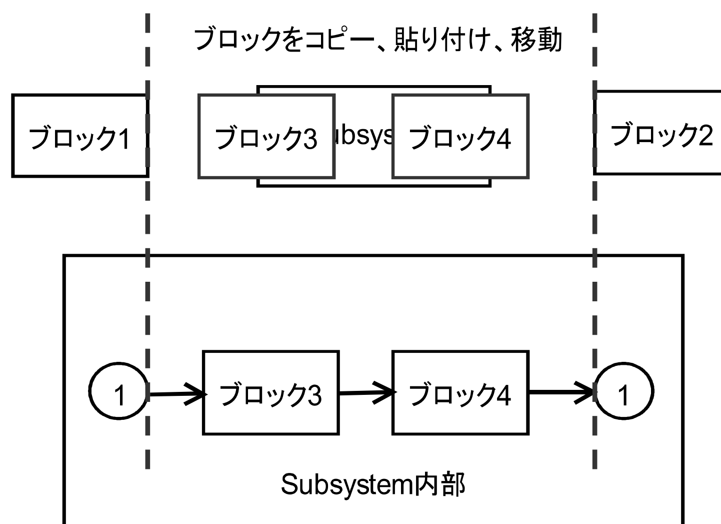


図 5.8: サブシステムブロックの展開の手順 3

手順 4：展開するサブシステムブロックを削除する.

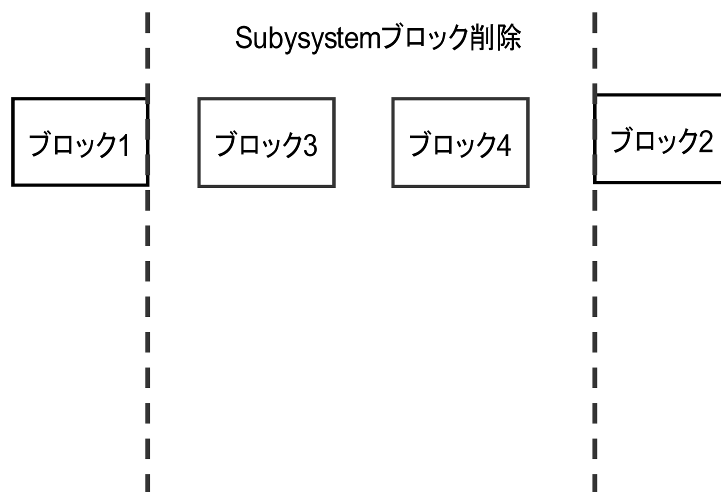


図 5.9: サブシステムブロックの展開の手順 4

手順 5 : ブロックを結線する

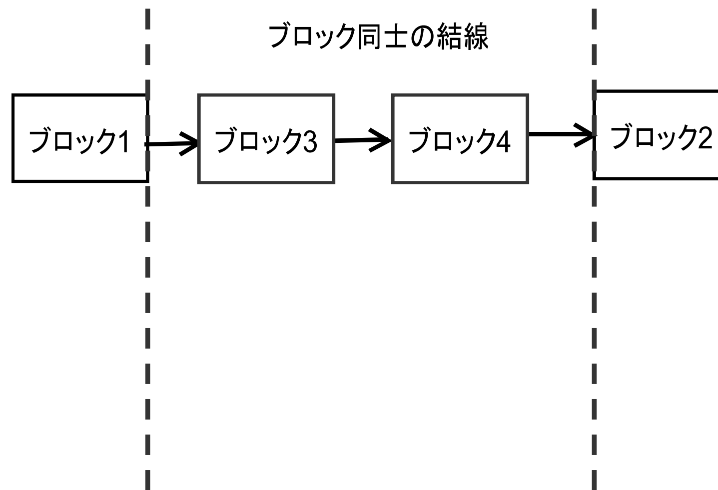


図 5.10: サブシステムブロックの展開の手順 5

[処理 4]　タスクグラフの行列と親情報の行列を作成する.

5.1.4 Matlab を用いた Simulink モデルの隣接情報抽出するプログラフの紹介

本プログラムは Matlab で作成されたスクリプト M-ファイル (付録 II の (1)):で実行されている. Simulink モデルに対してスケジューリングを行う際, Simulink モデルをタスクグラフとして扱う必要がある. Simulink モデル内部の各ブロックをタスクグラフのノードとして扱うために, 各ブロックに対してノード番号を割り当てる. 本プログラムを実行すると, 対象の Simulink モデルを構成する全てのブロック名と, 名前に対応したノード番号を順に割り当てた対応表 table.txt 及び, Simulink モデルの隣接関係を取得し, タスクグラフの隣接情報の形式で記述された result.txt の 2 つのテキストファイルを出力する.

図 5.11: 実行用プログラムの名前と保存場所

実行手順

本プログラムは abstract.m という名前のスクリプト M-ファイルで保存されている (図 5.11)．Matlab コマンドウィンドウ上でプログラム名を入力すると，

modelName = input(' モデル名を入力してください：','s')

を実行すると，隣接関係を取得する Simulink モデル名の入力を促す画面が表示される．Simulink モデル名を入力すると，

open_system(modelName)

により対象の Simulink モデルが開かる．ここでは，例として "sample7" というファイル名を入力する．そして，プログラムの実行結果である対応表とタスクグラフの隣接情報を保存した table.txt と result.txt の 2 つのテキストファイルが同一パス内に保存される．図 5.12 は本プログラムの実行画面を示す.

```
>> abstract
モデル名を入力してください：sample7
result.txtに保存しました。
table.txtに保存しました。
fx >> |
```

図 5.12: プログラムの実行画面

実行対象モデル

車両制御システム (付録 I) の実行用ファイルは model1.mdl である．車両制御システム内部には非常に多くのブロックが存在するため，車両制御システム全体の隣接情報を抽出し実行結果を検証するには煩雑と考えた．そこで，車両制御システム内部の簡単のモデルに対して本プログラムを実行し，その結果を示す．車両制御システムは複数のサブシステムを含み，さらに各サブシステム中でサブシステムが存在する場合がある．例えば，図 5.13 は Simulink モデルの ”model1 ”であり，着色したブロックはサブシステムブロック ”Coolant etc ”である．そして図はサブシステムブロック ”Coolant etc ”内部に含まれるサブシステムブロック ”CAL_O2 ”をさらに展開したものである．

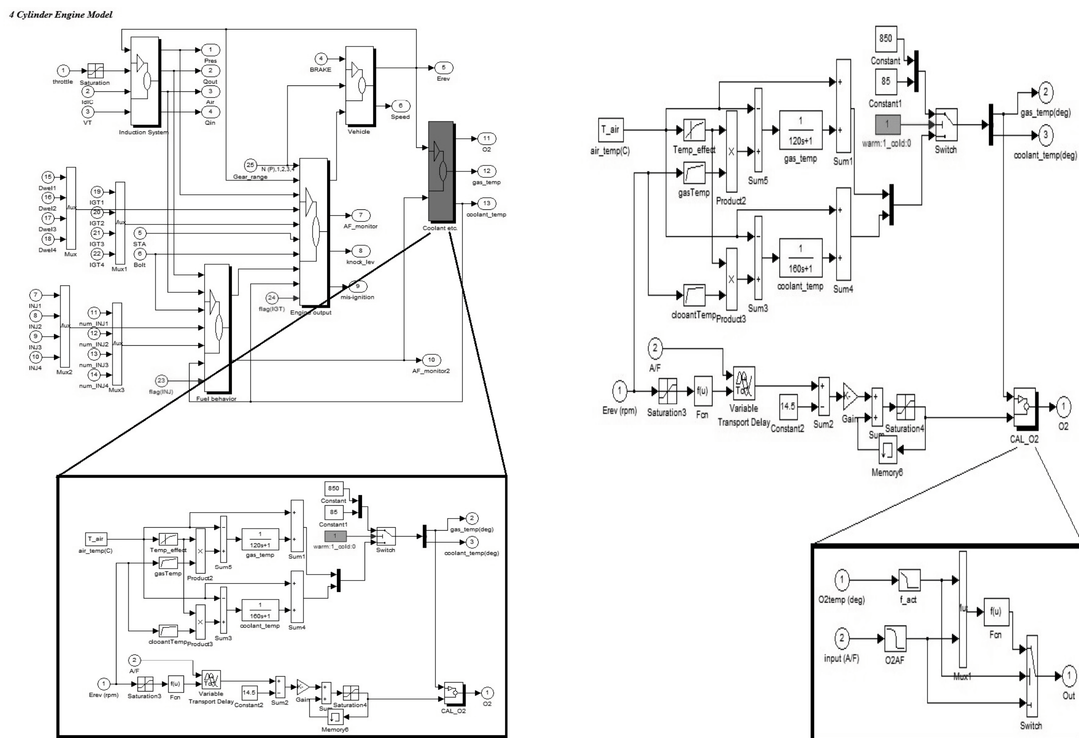


図 5.13: システム model1 中のサブシステムブロック Coolant etc

図 5.14: サブシステムブロック Coolant etc 中のサブシステムブロック CAL_O2

出力ファイル

次に，本プログラムを実行し，ファイル result.txt と table.txt に出力して，内容として図 5.16 と図 5.17 に示している．図 5.16 は例のサブシステム ”sample7 ” (図 5.15) のタスクグラフの行列である．図 5.17 は例のサブシステム ”sample7 ” のタスクグラフの対応表である．そのタスクグラフの対応表はブロック名とノード番号を含む．ブロック名は次の形式で表される．

モデル名/サブシステム名/サブシステム名/ $\cdots$ /ブロック名

例えば，図 5.14 のサブシステムブロック CAL_O2 中のあるブロックを表示する場合は「model1/Coolant etc/CAL_O2/ブロック名」のように表す．

図 5.16 と図 5.17 によってサブシステム ”sample7 ” (図 5.15) はタスクグラフに変更できる．その変更したタスクグラフのノード z_5 に例として説明する．ノード z_5 はサブシステム ”sample7 ” に対応したブロック名は sample7/Fcn3 である．そのノードの親は 1 つしかない，親のノード番号は 8 であり，ノード z_8 対応したサブシステム ”sample7 ” のブロック名が sample7/Mux1 である．

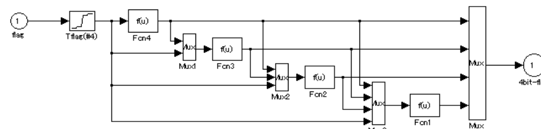


図 5.15: サブシステムブロック sample7

5.2 提案した方法のシミュレーションの実験と評価

ここでは，前章で紹介したブロック化法と再ブロック化法を評価するためのシミュレーション実験について述べる．今回のシミュレーション用タスクグラフは MATLAB/Simulink でモデル化される車両制御システム (付録 I) による変更されたタスクグラフである．このタスクグラフの総ノード数は 429 個である．提案した方法の

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

図 5.16: sample7 のタスクグラフの行列

1:SrcNode
2:sample7/flag↑
3:sample7/Fcn1
4:sample7/Fcn2
5:sample7/Fcn3
6:sample7/Fcn4
7:sample7/Mux
8:sample7/Mux1
9:sample7/Mux2
10:sample7/Mux3
11:sample7/Tflag(#4)
12:sample7/ 4bit-flag
13:SinkNode

図 5.17: sample7 のタスクグラフの対応表

有効性を確認するために，いくつかの実システムに対してシミュレーション実験を行った．タスクグラフのノードの実行時間とノード間の通信時間については，企業の共同研究者との協議で 0 または 1 と設定した．また，実行用のプロセッサの数は 2 つとする．

表 5.1 はシミュレーション結果を示しており，タスクグラフに 2 つプロセッサで実行して得られたタスクグラフの実行時間 (スケジュール結果) である．タスクグラフの実行時間の差はブロック化する前のタスクグラフの実行時間とブロック化した後のタスクグラフの実行時間との差であり，改善率 (%) は以下の式で計算した値である．

(タスクグラフの実行時間の差/ブロック化前のタスクグラフの実行時間)*100

表 5.1 の一行目に対して，490 は元のタスクグラフに 2 つのプロセッサで実行して得られたタスクグラフの実行時間である．240 はブロック化法で元のタスクグラフを処理して得られたタスクグラフに 2 つのプロセッサで実行して得られたタスクグラフの実行時間である．そしてブロック化前後の差は 250 である．表 5.1 の二行目に対して，240 はブロック化法で元のタスクグラフを処理して得られたタスクグラフに，2 つのプロセッサで実行して得られたタスクグラフの実行時間である．219 はブロック化法で処理したタスクグラフにもう一度再ブロック化法で処理して得られたタスクグラフに，2 つのプロセッサで実行して得られたタスクグラフの実行時間である．そしてブロック化前後の差は 21 である．

表 5.1 を見て分かるように，(1) ブロック化法を利用してスケジュール結果が 51%に減少した；(2) 再ブロック化法を利用してスケジュール結果がさらに 8.8%に減少し

た．この結果に見て分かるように，実システムに対して提案したブロック化法と再ブロック化法は有効性を示している．しかし，再ブロック化法は実システムに対して顕著な改善率の変化は見られなかった．このような結果になる理由として，(1) 実システムにブロック化法で処理する時に多くのサブグラフがブロックに変更した．ブロックが増えると実システムの複雑さが増す；(2) 任意のタスクグラフに対してブロック化の限界があり，ブロック化法で処理したタスクグラフに対してブロックできるノードが少なくなった．その他，提案したブロック化法と再ブロック化法は実システムに 55.3% ((490-219)/490 × 100) の改善率を与えた．この結果によって提案したブロック化法と再ブロック化法は大規模な実際のシステムに有効である．

表 5.1: 実システムにブロック化法と再ブロック化法に基づいてスケジュール結果

提案する方法	タスクグラフの実行時間 (ns)		実行前後の差	改善率
	実行前	実行後	(ns)	(%)
ブロック化法	490	240	250	51
再ブロック化法	240	219	21	8.8

5.3 Matlab プログラミング言語に基づくブロック化の実現

実行用プログラムはブロック化法のプログラム block_construction.m(付録 II の (2)) と再ブロック化法のプログラム block_reconstruction.m(付録 II の (3)) とスケジュール結果を求めるプログラム Scheduling_1.m(付録 II の (4)) と最長パスの長さを求めるプログラム CP.m(付録 II の (5)) 4 つがある．そしてブロック化法と再ブロック化法のプログラムには複数のサブプログラム（関数）が存在する．Matlab の実行環境でサブプログラムが単独で実行できる．

車両制御システム全体にプログラムを使用するには煩雑と考えた．ここでは車両

制御システム「model1」中の展開していない Coolant Control サブシステムから変更したタスクグラフ例としてそれらのサブプログラムを一つずつ説明し，実行結果を出す.

5.3.1 coolant タスクグラフについて

実行用 *txt* ファイルと実行用プログラムが同じファイルに保存する (図 5.18)．図 5.19， 5.20 は coolant タスクグラフとその行列.

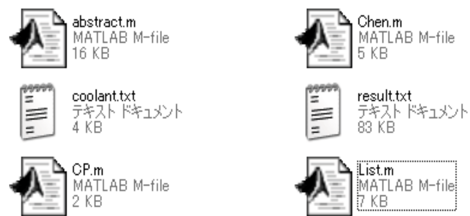


図 5.18: coolant タスクグラフの txt ファイル

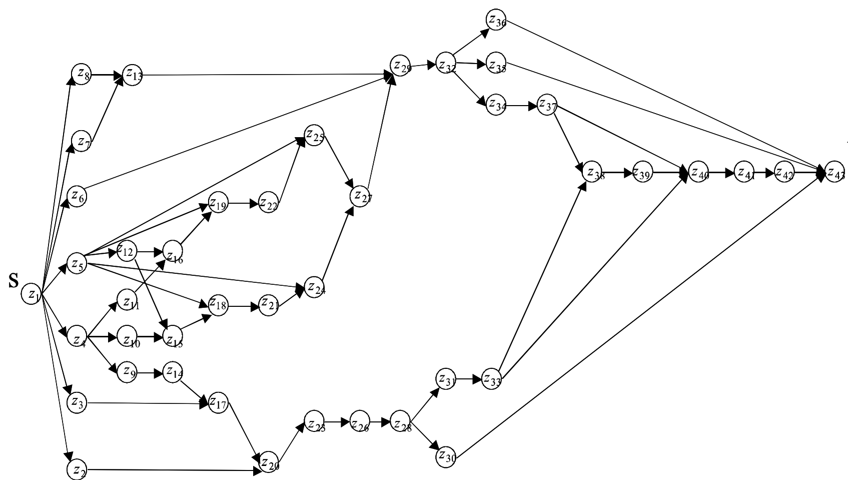


図 5.19: coolant タスクグラフ

41										
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0	0	0	0
3	1	1	1	0	0	0	0	0	0	0
4	1	1	1	0	0	0	0	0	0	0
5	1	1	1	0	0	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	1	1	1	0	0	0	0	0	0	0
8	1	1	1	0	0	0	0	0	0	0
9	1	1	4	1	0	0	0	0	0	0
10	1	1	4	1	0	0	0	0	0	0
11	1	1	4	1	0	0	0	0	0	0
12	1	1	5	1	0	0	0	0	0	0
13	1	2	7	1	8	1	0	0	0	0
14	1	1	9	1	0	0	0	0	0	0
15	1	2	10	1	12	1	0	0	0	0
16	1	2	11	1	12	1	0	0	0	0
17	1	2	3	1	14	1	0	0	0	0
18	1	2	15	1	5	1	0	0	0	0
19	1	2	16	1	5	1	0	0	0	0
20	1	2	2	1	17	1	0	0	0	0
21	1	1	18	1	0	0	0	0	0	0
22	1	1	19	1	0	0	0	0	0	0
23	1	1	20	1	0	0	0	0	0	0
24	1	2	21	1	5	1	0	0	0	0
25	1	2	22	1	5	1	0	0	0	0
26	1	1	23	1	0	0	0	0	0	0
27	1	2	24	1	25	1	0	0	0	0
28	1	1	26	1	0	0	0	0	0	0
29	1	3	27	1	6	1	13	1	0	0
30	1	1	28	1	0	0	0	0	0	0
31	1	1	28	1	0	0	0	0	0	0
32	1	1	29	1	0	0	0	0	0	0
33	1	1	31	1	0	0	0	0	0	0
34	1	1	32	1	0	0	0	0	0	0
35	1	1	32	1	0	0	0	0	0	0
36	1	1	32	1	0	0	0	0	0	0
37	1	1	34	1	0	0	0	0	0	0
38	1	2	33	1	37	1	0	0	0	0
39	1	1	38	1	0	0	0	0	0	0
40	1	3	33	1	37	1	39	1	0	0
41	1	1	40	1	0	0	0	0	0	0
42	1	1	41	1	0	0	0	0	0	0
43	0	4	30	0	42	0	35	0	36	0

図 5.20: coolant タスクグラフの行列

5.3.2 データを読み込むサブプログラム (Chen.m)

まず，タスクグラフデータを読み込み，ファイル名「coolant.txt」を入力する．図 5.21 は Matlab のコマンドウィンドウ中で表示の実行結果の一部分である．

```
>> Chen
taskgraphの名前を入力してください：coolant.txt
     1     0     0     0     0     0     0     0     0     0     0
     2     1     1     1     0     0     0     0     0     0     0
     3     1     1     1     0     0     0     0     0     0     0
     4     1     1     1     0     0     0     0     0     0     0
     5     1     1     1     0     0     0     0     0     0     0
     6     1     1     1     0     0     0     0     0     0     0
     7     1     1     1     0     0     0     0     0     0     0
     8     1     1     1     0     0     0     0     0     0     0
     9     1     1     4     1     0     0     0     0     0     0
    10     1     1     4     1     0     0     0     0     0     0
    11     1     1     4     1     0     0     0     0     0     0
    12     1     1     5     1     0     0     0     0     0     0
    13     1     2     7     1     8     1     0     0     0     0
    14     1     1     9     1     0     0     0     0     0     0
    15     1     2    10     1    12     1     0     0     0     0
    16     1     2    11     1    12     1     0     0     0     0
    17     1     2     3     1    14     1     0     0     0     0
    18     1     2    15     1     5     1     0     0     0     0
    19     1     2    16     1     5     1     0     0     0     0
    20     1     2     2     1    17     1     0     0     0     0
```

図 5.21: ファイルからの読み込み

データの意味は順番にノードの番号，ノードの実行時間，ノードの親の数，ノードの親の番号，親からの通信時間，ノードの親の番号，親からの通信時間 $\cdots$ となっていく．

次はタスクグラフと各ノードの情報構造体を作る．プログラムを実行すると，Matlab のコマンドウィンドウ中で各ノードの情報構造体とタスクグラフの構造体を表示する．ここでは，ノード z_2 を例として説明する．図 5.22 はノード z_2 の情報構造体と coolant タスクグラフの構造体を示す．

```
      task_number: 2
          exetime: 1
               ip: 1
        ip_number: 1
          ip_read: 0
               is: 1
   sum_write_time: 1
    sum_read_time: 0
        is_number: 20
```

```
     total_task: 41
           task: [1x43 struct]
       total_is: 60
       total_ip: 60

>>
```

図 5.22: ノード z_2 とタスクグラフの構造体

図 5.22 の左はノード z_2 の構造体，右は coolant タスクグラフの構造体である．ノードの構造体について，task_number はノードの番号，exetime はノードの実行時間，ip はノードの親の数，ip_number は親の番号，ip_read は親からの通信時間，is はノードの子の数，sum_write_time は総書き出し時間，sum_read_time は総読み込み時間，is_number は子の番号である．タスクグラフの構造体について，total_task は総タスクの数，task は各ノード番号を示す，total_is はすべての子の数，total_ip はすべての親の数である．

5.3.3 各ノードに最早最遅開始時刻と優先リストを求めるサブプログラム (List.m)

このサブプログラムを実行し続けると，各ノードの構造体の中に最早開始時刻と最遅開始時刻を追加できる．それだけではなくて，タスクグラフの構造体の中に各ノードの優先度を表す優先リストを追加できる．ここではまたノード z_2 を例として説明する．図 5.23 は最早最遅開始時刻の情報を追加したノード z_2 の構造体と優先リストの情報を追加した coolant タスクグラフの構造体を示す．

```
     task_number: 2
         exetime: 1
              ip: 1
       ip_number: 1
         ip_read: 0
              is: 1
  sum_write_time: 1
   sum_read_time: 0
       is_number: 20
        earliest: 0
          latest: 23
```

```
   total_task: 41
         task: [1x43 struct]
     total_is: 60
     total_ip: 60
priority_list: [5 4 12 11 10 15 16 18 19 9 14 21 22 3 17 24 25 8 2 7 13 20 27 6 23 29 26 32 28 31 34 33 37 38 39 40 41 36 30 35 42]
        value: [0 3 6 7 7 10 10 14 14 15 18 18 18 19 21 21 21 23 23 23 25 25 25 27 29 29 32 34 35 39 39 42 42 46 50 53 58 61 61 61 61]

>>
```

図 5.23: 最早最遅開始時刻と優先リストの情報を追加したノード z_2 とタスクグラフの構造体

図 5.23 の上はノード z_2 の構造体，下は coolant タスクグラフの構造体である．ここでは追加した部分だけを説明する．ノード z_2 の構造体の中で earliest はこのノードの最早開始時刻，latest はこのノードの最遅開始時刻である．ノード z_2 の構造体

の中で2つの情報を追加した．タスクグラフの構造体の中で priority_list はノードの優先リストである．それによって各ノードの実行順位は「z_5, z_4, z_{12}, z_{11}, z_{10}, z_{15}, z_{16}, $z_{18} \cdots$」．value は各ノードの最遅開始時刻であり，それらの値に基づいて各ノード優先度を判断する．値は小さいほうが優先度が高く，値は大きいほうが優先度が低い．

5.3.4 クリティカルパス法（CP 法）というシミュレーション法を求めるプログラム (CP.m)

このプログラムは提案したブロック化法の専用のシミュレーション法である．その理由はブロック化法に含まれる基本ブロック化法と追加ブロック化法が評価するために無限のプロセッサを利用して実行することである．このプログラムを利用してブロック化法で実行されたタスクグラフにどのぐらい通信時間を減らしたかを評価できる．

```
CP長： 47
1 の srlen： 47
2 の srlen： 28
3 の srlen： 31
4 の srlen： 44
5 の srlen： 47
6 の srlen： 26
7 の srlen： 29
8 の srlen： 29
9 の srlen： 34
10 の srlen： 41
11 の srlen： 41
12 の srlen： 42
13 の srlen： 28
14 の srlen： 32
15 の srlen： 39
16 の srlen： 39
17 の srlen： 30
18 の srlen： 36
19 の srlen： 36
20 の srlen： 27
21 の srlen： 33
22 の srlen： 33
23 の srlen： 24
24 の srlen： 31
25 の srlen： 31
26 の srlen： 22
27 の srlen： 28
28 の srlen： 20
29 の srlen： 25
30 の srlen： 1
31 の srlen： 17
32 の srlen： 21
```

```
CP長： 18
1 の srlen： 18
2 の srlen： 0
3 の srlen： 5
4 の srlen： 0
5 の srlen： 18
6 の srlen： 11
7 の srlen： 0
8 の srlen： 11
9 の srlen： 0
10 の srlen： 15
11 の srlen： 0
12 の srlen： 0
13 の srlen： 0
14 の srlen： 0
15 の srlen： 0
16 の srlen： 0
17 の srlen： 0
18 の srlen： 0
19 の srlen： 0
20 の srlen： 0
21 の srlen： 0
22 の srlen： 0
23 の srlen： 0
24 の srlen： 0
25 の srlen： 0
26 の srlen： 0
27 の srlen： 13
28 の srlen： 0
29 の srlen： 10
30 の srlen： 1
31 の srlen： 0
32 の srlen： 0
```

図 5.24: ブロック化法で実行したタスクグラフ前後の最長パスの長さ

図 5.24 の左はブロック化法で，実行前の各ノードの最長パスの長さの一部分，右はブロック化法で，実行後の各ノードの最長パスの長さの一部分である．見てわかるように，左の方はノード z_1（ソースノード）の最長パスの長さが 47 だから，このタスクグラフの最長パスの長さが 47，一方，右の方はノード z_1 の最長パスの長さが 18 だから，このタスクグラフの最長パスの長さが 18 である，ブロック化法で実行され

たタスクグラフの通信時間が 29 減少した．結果として 3 章の最後文章に載せた．

5.3.5 各ノードの先祖と子孫を求めるサブプログラム (Predecessor.m, Successor.m)

各ノードの先祖を求めるサブプログラム (Predecessor.m)

このサブプログラムを実行し続けると，各ノードの構造体の中に先祖の情報を追加できる．ここではまたノード z_2 を例として説明する．図 5.25 はノード z_2 の先祖の情報を追加したノード z_2 の構造体である．そして pre はノード z_2 の先祖の数，pre_number はノード z_2 の先祖の番号である．

```
    task_number: 2
        exetime: 1
             ip: 1
      ip_number: 1
        ip_read: 0
             is: 1
 sum_write_time: 1
  sum_read_time: 0
      is_number: 20
       earliest: 0
         latest: 23
            pre: 1
     pre_number: 1
```

図 5.25: 先祖の情報を追加したノード z_2 の構造体

各ノードの子孫を求めるサブプログラム (Successor.m)

このサブプログラムを実行し続けると，各ノードの構造体の中に子孫の情報を追加できる．ここではまたノード z_2 を例として説明する．図 5.26 はノード z_2 の子孫の情報を追加したノード z_2 の構造体である．そして suc はノード z_2 の子孫の数，suc_number はノード z_2 の子孫の番号である．

```
    task_number: 2
        exetime: 1
             ip: 1
      ip_number: 1
        ip_read: 0
             is: 1
 sum_write_time: 1
  sum_read_time: 0
      is_number: 20
       earliest: 0
         latest: 23
            pre: 1
     pre_number: 1
            suc: 13
     suc_number: [20 23 26 28 30 31 33 38 39 40 41 42 43]
```

図 5.26: 子孫の情報を追加したノード z_2 の構造体

5.3.6 ブロック化法の基本ブロック化法のサブプログラム (Block.m)

このサブプログラムは 1 回だけ実行する．実行したタスクグラフは基本ブロック化法の 4 つパターンが存在しない．

まず，基本ブロック化法の 4 つパターンの条件を満たし，ブロック化できる親子関係のノードを求める．実行結果は図 5.27 のように示している．例えば，ノード z_5 と z_{12} はブロック化できる親子関係のノードである．

5	12
9	14
15	18
16	19
18	21
19	22
20	23
21	24
22	25
23	26
26	28
29	32
31	33
34	37
38	39
39	40
40	41
41	42

図 5.27: 基本ブロック化法（1）: ブロック化できる親子関係のノード

上のブロック化できる親子関係のノードの中で，ある親子関係のノードが重複で表したので，次はその重複した親子関係のノードを組み合わせる必要がある．実行結果は図 5.28 のように示している．例えば，ブロック z_{15}，z_{18} とブロック z_{18}，z_{21} とブロック z_{21}，z_{24} が重複したので，組み合わせた後で，1 つのブロックに z_{15} と z_{18} と z_{21} と z_{24} を含む．

<<基本ブロック化法>>で得られたブロック：

5	12	0	0	0
9	14	0	0	0
15	18	21	24	0
16	19	22	25	0
20	23	26	28	0
29	32	0	0	0
31	33	0	0	0
34	37	0	0	0
38	39	40	41	42

ブロックの総数：（9）

図 5.28: 基本ブロック化法（2）: 重複した親子関係のノードの組み合わせ

次は各ブロックが新たなノードに変更することである．新たなノードの番号はブ

ロックに含まれる一番親の番号とする．新たなノードの親と子はブロックに含まれるすべてのノードの親と子，新たなノードの実行時間はブロックに含まれる各ノードの実行時間の総和である．図 5.29 は基本ブロック化法で実行されたタスクグラフの行列である．

1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0	0	0	0
3	1	1	1	0	0	0	0	0	0	0
4	1	1	1	0	0	0	0	0	0	0
5	2	1	1	0	0	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	1	1	1	0	0	0	0	0	0	0
8	1	1	1	0	0	0	0	0	0	0
9	2	1	4	1	0	0	0	0	0	0
10	1	1	4	1	0	0	0	0	0	0
11	1	1	4	1	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0
13	1	2	7	1	8	1	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	4	2	5	1	10	1	0	0	0	0
16	4	2	5	1	11	1	0	0	0	0
17	1	2	3	1	9	1	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0
20	4	2	2	1	17	1	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0
27	1	2	15	1	16	1	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0
29	2	3	6	1	13	1	27	1	0	0
30	1	1	20	1	0	0	0	0	0	0
31	2	1	20	1	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0	0	0
34	2	1	29	1	0	0	0	0	0	0
35	1	1	29	1	0	0	0	0	0	0
36	1	1	29	1	0	0	0	0	0	0
37	0	0	0	0	0	0	0	0	0	0
38	5	2	31	1	34	1	0	0	0	0
39	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0
41	0	0	0	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0	0	0	0
43	0	4	30	0	35	0	36	0	38	0

図 5.29: 基本ブロック化法で実行して得られたタスクグラフの行列

実行結果を見て分かるように，元のタスクグラフの行列 (図 5.20) と比べると，あるノードの実行時間＞「１」になった，そしてあるノードのすべての情報が「0」になった．それは複数のノードがブロックして新たなノードに変更したからである．例えば，ノード z_5,z_{12} を含めるブロックが新たなノードに変更すると，ノード z_5 はノード z_{12} の親であるので，新たなノードの番号は５番である．そして新たなノー

ド z_5 の実行時間は元のノード z_5 と z_{12} の実行時間の総和，2 である．新たなノード z_5 の親の情報も元のノード z_5 と z_{12} の情報である．最後，ノード z_{12} が存在しないので，すべての情報が「0」にする．

5.3.7 ブロック化法の追加ブロック化法のサブプログラム(Newblock.m)

基本ブロック化法で実行されたタスクグラフを利用して追加ブロック化法で実行し続ける．そして追加ブロック化法のサブプログラムは何回も実行することが必要である．それは，このサブプログラムは実行すると最初に基本ブロック化法と同じように，複数のブロックを組み合わせることができるからである．しかし，すべてのブロックが一気に新たなノードに変更したら，変更したタスクグラフ中に閉路が存在する可能性がある．だから，すべてのブロックが一気に新たなノードに変更するのではなくて，１つのブロックが変更する時にまずタスクグラフ中に閉路を構成できるかどうかをチェックし，構成できなければブロックが新たなノードに変更してタスクグラフを変更する．　このサブプログラムはタスクグラフ中に閉路が構成できないブロックが存在する限り実行し続ける．

```
<<追加ブロック化法で>>で得られたブロック：
     7    13     0     0
    10    15     0     0
    11    16     0     0
     9    17    20    31
    27    29    34    38

現在のタスクグラフのブロック総数：（5）
新たなブロックの番号は(  7  ).
そのブロックに含まれるノード:
7 13
```

```
<<追加ブロック化法で>>で得られたブロック：
     8     7     0     0
    10    15     0     0
    11    16     0     0
     9    17    20    31
    27    29    34    38

現在のタスクグラフのブロック総数：（5）
新たなブロックの番号は(  8  ).
そのブロックに含まれるノード:
8 7
```

図 5.30: 追加ブロック化法１回目と２回目の実行結果

図 5.30 は基本ブロック化法で実行されたタスクグラフに追加ブロック化法で実行した最初の２回の実行結果である．図 5.30 の左は１回目の実行結果であり，追加ブロック化法で実行して５つのブロックが構成したが，ノード z_7 と z_{13} を含まれる１番目のブロックを選んで閉路が構成できないことを確認し，そしてこのブロックが新たなノードに変更し，番号は７である．それからタスクグラフを変更し，もう一度追加ブロック化法を実行して，図 5.30 の右のように示した．ノード z_8 と新たな

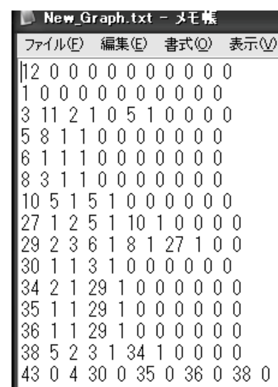
ノード z_7 に含まれたブロックを，新たなノードに変更し，タスクグラフを変更して，追加ブロック化法で実行し続ける．

```
現在のタスクグラフのブロック総数： （1）
ブロック化できないブロックの確認
     5    10    27    29    34    38

★★★★★このタスクグラフにブロック化法できない★★★
★★★実行できるブロックの数は"0"ので，プログラムそのまま終了★★★
追加ブロック化法の実行回数：（ 8 ）回.
```

図 5.31: 追加ブロック化法最後の 1 回目の実行結果

追加ブロック化法の結果として，図 5.31 のように追加ブロック化法は 8 回実行した．そして 1 つの閉路を構成できるブロックを発見した．最後のブロック化法の結果は Result_Graph.txt と New_Graph.txt2 つのファイルに出力する．

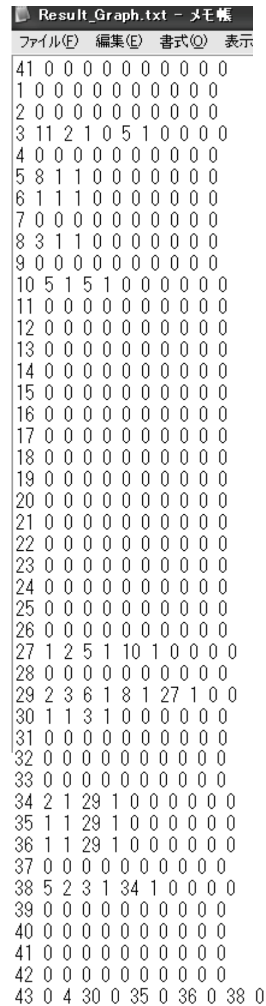


New_Graph.txt - メモ帳
ファイル(F) 編集(E) 書式(O) 表示(V)

```
12 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
3 11 2 1 0 5 1 0 0 0 0
5 8 1 1 0 0 0 0 0 0 0
6 1 1 1 0 0 0 0 0 0 0
8 3 1 1 0 0 0 0 0 0 0
10 5 1 5 1 0 0 0 0 0 0
27 1 2 5 1 10 1 0 0 0 0
29 2 3 6 1 8 1 27 1 0 0
30 1 1 3 1 0 0 0 0 0 0
34 2 1 29 1 0 0 0 0 0 0
35 1 1 29 1 0 0 0 0 0 0
36 1 1 29 1 0 0 0 0 0 0
38 5 2 3 1 34 1 0 0 0 0
43 0 4 30 0 35 0 36 0 38 0
```

図 5.32: New_Graph.txt ファイル

New_Graph.txt ファイルの中身は図 5.32 のように示している．タスクグラフ中の各ノードに対して，情報がないノードを削除し，情報があるノードは残している．このファイルは実行ファイルではなくて，最終の結果をチェックするファイルである．



Result_Graph.txt - メモ帳

ファイル(F) 編集(E) 書式(O) 表示

```
41 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0
3 11 2 1 0 5 1 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0
5 8 1 1 0 0 0 0 0 0 0
6 1 1 1 0 0 0 0 0 0 0
7 0 0 0 0 0 0 0 0 0 0
8 3 1 1 0 0 0 0 0 0 0
9 0 0 0 0 0 0 0 0 0 0
10 5 1 5 1 0 0 0 0 0 0
11 0 0 0 0 0 0 0 0 0 0
12 0 0 0 0 0 0 0 0 0 0
13 0 0 0 0 0 0 0 0 0 0
14 0 0 0 0 0 0 0 0 0 0
15 0 0 0 0 0 0 0 0 0 0
16 0 0 0 0 0 0 0 0 0 0
17 0 0 0 0 0 0 0 0 0 0
18 0 0 0 0 0 0 0 0 0 0
19 0 0 0 0 0 0 0 0 0 0
20 0 0 0 0 0 0 0 0 0 0
21 0 0 0 0 0 0 0 0 0 0
22 0 0 0 0 0 0 0 0 0 0
23 0 0 0 0 0 0 0 0 0 0
24 0 0 0 0 0 0 0 0 0 0
25 0 0 0 0 0 0 0 0 0 0
26 0 0 0 0 0 0 0 0 0 0
27 1 2 5 1 10 1 0 0 0 0
28 0 0 0 0 0 0 0 0 0 0
29 2 3 6 1 8 1 27 1 0 0
30 1 1 3 1 0 0 0 0 0 0
31 0 0 0 0 0 0 0 0 0 0
32 0 0 0 0 0 0 0 0 0 0
33 0 0 0 0 0 0 0 0 0 0
34 2 1 29 1 0 0 0 0 0 0
35 1 1 29 1 0 0 0 0 0 0
36 1 1 29 1 0 0 0 0 0 0
37 0 0 0 0 0 0 0 0 0 0
38 5 2 3 1 34 1 0 0 0 0
39 0 0 0 0 0 0 0 0 0 0
40 0 0 0 0 0 0 0 0 0 0
41 0 0 0 0 0 0 0 0 0 0
42 0 0 0 0 0 0 0 0 0 0
43 0 4 30 0 35 0 36 0 38 0
```

図 5.33: 追加ブロック化法最後の実行結果

Result_Graph.txt ファイルの中身は図 5.33 のように示している．タスクグラフ中の各ノードに対して，情報がないノード或いは情報があるノードがすべて残っている．このファイルは実行ファイルである．後で再ブロック化法の実行タスクグラフとして利用する．

5.3.8 ブロック化前後の対応表を求めるサブプログラム（blocktable.m）

blocktable.m は最後は得られたタスクグラフのノードと元のタスクグラフのノード間の対応表を作るプログラムである．その結果は図 5.34 のように示す.

```
ブロック<1>から変更したノード番号は 5 です.
対応するノード番号は：
     4     5    11    12    16    19    22    25

ブロック<2>から変更したノード番号は 3 です.
対応するノード番号は：
     2     3     9    14    17    20    23    26    28    31    33

ブロック<3>から変更したノード番号は 10 です.
対応するノード番号は：
    10    15    18    21    24

ブロック<4>から変更したノード番号は 29 です.
対応するノード番号は：
    29    32

ブロック<5>から変更したノード番号は 34 です.
対応するノード番号は：
    34    37

ブロック<6>から変更したノード番号は 38 です.
対応するノード番号は：
    38    39    40    41    42

ブロック<7>から変更したノード番号は 8 です.
対応するノード番号は：
     7     8    13

>>
```

図 5.34: ブロック化前後の対応結果

その対応表によってブロック化前後の対応結果がはっきり見えるブロックの総数が 7 個があり，そして変更したタスクグラフのノード z_5 に対して対応した元のタスクグラフのノードは z_4,z_5,z_{11},z_{12},z_{16},z_{19},z_{22},z_{25} である.

5.3.9 評価用スケジューリング法のプログラム（Scheduling_1.m）

このプログラムを利用して，元のタスクグラフのスケジュール結果またはブロック化法で実行されたタスクグラフのスケジュール結果を求められる．

N番号	P番号	endT	読込	実行	書出
0	0	0	0	0	0
2	1	32	32	33	34
3	1	24	24	25	26
4	2	0	0	1	4
5	1	0	0	1	6
6	2	38	38	39	40
7	2	32	32	33	34
8	1	30	30	31	32
9	1	18	19	20	21
10	2	7	8	9	10
11	2	4	5	6	7
12	1	6	7	8	10
13	1	34	36	37	38
14	1	21	22	23	24
15	1	10	12	13	14
16	2	10	12	13	14
17	1	26	28	29	30
18	1	14	16	17	18
19	2	14	16	17	18
20	2	34	36	37	38
21	2	18	19	20	21
22	2	21	22	23	24
23	2	40	41	42	43
24	2	24	26	27	28
25	2	28	30	31	32
26	2	43	44	45	46
27	1	38	40	41	42
28	2	46	47	48	50
29	1	42	45	46	47
30	2	59	60	61	61
31	2	50	51	52	53
32	1	47	48	49	52
33	2	53	54	55	57
34	1	52	53	54	55
35	2	61	62	63	63
36	2	57	58	59	59
37	1	55	56	57	59
38	1	59	61	62	63
39	1	63	64	65	66
40	1	66	69	70	71
41	1	71	72	73	74
42	1	74	75	76	76
0	0	0	0	0	0
76					

N番号	P番号	endT	読込	実行	書出
0	0	0	0	0	0
2	2	6	6	6	6
3	2	11	12	23	25
4	2	6	6	6	6
5	1	0	0	8	11
6	2	4	4	5	6
7	2	6	6	6	6
8	2	0	0	3	4
9	2	6	6	6	6
10	1	11	12	17	18
11	2	6	6	6	6
12	2	6	6	6	6
13	2	6	6	6	6
14	2	6	6	6	6
15	2	6	6	6	6
16	2	6	6	6	6
17	2	6	6	6	6
18	2	6	6	6	6
19	2	6	6	6	6
20	2	6	6	6	6
21	2	6	6	6	6
22	2	6	6	6	6
23	2	6	6	6	6
24	2	6	6	6	6
25	2	6	6	6	6
26	2	6	6	6	6
27	1	18	20	21	22
28	2	6	6	6	6
29	1	22	25	27	30
30	2	25	26	27	27
31	2	6	6	6	6
32	2	6	6	6	6
33	2	6	6	6	6
34	1	30	31	33	34
35	2	30	31	32	32
36	2	32	33	34	34
37	2	6	6	6	6
38	1	34	36	41	41
39	2	6	6	6	6
40	2	6	6	6	6
41	2	6	6	6	6
42	2	6	6	6	6
0	0	0	0	0	0
41					

図 5.35: ブロック化法で実行前後のタスクグラフのスケジュール結果

図 5.35 の左は元の coolant タスクグラフのスケジュール結果，右はブロック化法で実行された coolant タスクグラフのスケジュール結果である．実行用プロセッサの数は２つである．

タスクグラフのスケジュール結果について,「N 番号」はノードの番号,「P 番号」は実行するプロセッサの番号,「endT」はノードの実行開始の時刻,「読込」はノードの親から読み込み時間を追加した時刻,「実行」はノードの実行時間を追加した時刻,「書出」はノードの子への書き出し時間を追加した時刻である．そして各スケジュール結果の一番下の単一の数字，このタスクグラフのスケジュール結果（タスクグラフの実行時間）である．元の coolant タスクグラフのスケジュール結果は 76，ブロック化法で実行されたタスクグラフのスケジュール結果は 41 である..

5.3.10 再ブロック化法の専用スケジューリング法 (LST-EST 法) のサブプログラム（Scheduling.m）

このサブプログラムを利用してブロック化法で実行されたタスクグラフに LST-EST 法で再ブロック法専用のスケジュール結果を求める．このサブプログラムと再ブロック化法のサブプログラムを何回も実行し続ける．

N番号	P番号	endT	読込	実行	書出
0	0	0	0	0	0
2	2	6	6	6	6
3	2	11	12	23	25
4	2	6	6	6	6
5	1	0	0	8	11
6	2	4	4	5	6
7	2	6	6	6	6
8	2	0	0	3	4
9	2	6	6	6	6
10	1	11	12	17	18
11	2	6	6	6	6
12	2	6	6	6	6
13	2	6	6	6	6
14	2	6	6	6	6
15	2	6	6	6	6
16	2	6	6	6	6
17	2	6	6	6	6
18	2	6	6	6	6
19	2	6	6	6	6
20	2	6	6	6	6
21	2	6	6	6	6
22	2	6	6	6	6
23	2	6	6	6	6
24	2	6	6	6	6
25	2	6	6	6	6
26	2	6	6	6	6
27	1	18	20	21	22
28	2	6	6	6	6
29	1	22	25	27	30
30	2	30	31	32	32
31	2	6	6	6	6
32	2	6	6	6	6
33	2	6	6	6	6
34	1	30	31	33	34
35	2	32	33	34	34
36	2	34	35	36	36
37	2	6	6	6	6
38	1	34	36	41	41
39	2	6	6	6	6
40	2	6	6	6	6
41	2	6	6	6	6
42	2	36	36	36	36
0	0	0	0	0	0
41					

N番号	P番号	endT	読込	実行	書出
0	0	0	0	0	0
2	2	6	6	6	6
3	1	15	16	28	29
4	2	6	6	6	6
5	1	0	0	13	15
6	2	4	4	5	6
7	2	6	6	6	6
8	2	0	0	3	4
9	2	6	6	6	6
10	2	6	6	6	6
11	2	6	6	6	6
12	2	6	6	6	6
13	2	6	6	6	6
14	2	6	6	6	6
15	2	6	6	6	6
16	2	6	6	6	6
17	2	6	6	6	6
18	2	6	6	6	6
19	2	6	6	6	6
20	2	6	6	6	6
21	2	6	6	6	6
22	2	6	6	6	6
23	2	6	6	6	6
24	2	6	6	6	6
25	2	6	6	6	6
26	2	6	6	6	6
27	2	15	18	21	24
28	2	6	6	6	6
29	2	6	6	6	6
30	2	6	6	6	6
31	2	6	6	6	6
32	2	6	6	6	6
33	2	6	6	6	6
34	1	29	31	38	38
35	2	29	30	31	31
36	2	31	32	33	33
37	2	6	6	6	6
38	2	6	6	6	6
39	2	6	6	6	6
40	2	6	6	6	6
41	2	6	6	6	6
42	2	33	33	33	33
0	0	0	0	0	0
38					

図 5.36: 再ブロック化法で実行前後のタスクグラフのスケジュール結果

図 5.36 の左はブロック化法で実行されたタスクグラフに LST-EST 法で求めたス

ケジュール結果，右はブロック化法で実行されたタスクグラフにもう一度再ブロック化法を実行したスケジュール結果である．実行用プロセッサの数は２つである

ここではLST-EST 法で求めたタスクグラフのスケジュール結果と普通のスケジューリング法で求めたタスクグラフのスケジュール結果が同じである．両方とも 41 である．そしてもう一度再ブロック化法で実行するとスケジュール結果が 38 になった．

図 5.36 のような表におり，タスクグラフのスケジューリングを表すことができる．図の左右のタスクグラフのスケジュール結果に基づいて 4 章の図 ??と図 ??ようなスケジューリングを表すことができる．

5.3.11　再ブロック化法のサブプログラム（ReBlock.m）

再ブロック化法のサブプログラムは何回も実行することが必要である．

まずは再ブロック化法の 3 つの条件を満たしたスケジューリング上の隣接のノードパターンを求める．

次は求めたパターンが重複しないようにチェックする．例えば，スケジューリング上のノード z_2 と z_3 は条件を満たす隣接ノードパターンでも，ノード z_3 と z_4 も条件を満たす隣接ノードパターンである．この 2 つのパターンの中でノード z_3 が重複したので，求めたパターンの順位によってノード z_3 と z_4 のパターンを削除し，ノード z_2 と z_3 のパターンを残す．

最後には求めた各隣接のノードパターンはブロックして新たなノードに変更してタスクグラフを全体に変更する．

```
ブロックに含まれるノード:

block =

     5    10
    27    29
    34    38
     3    30


total_block =

     4
```

図 5.37: 再ブロック化法で実行した結果

```
graph =

     1     0     0     0     0     0     0     0     0     0     0
     2     0     0     0     0     0     0     0     0     0     0
     3    12     2     1     0     5     1     0     0     0     0
     4     0     0     0     0     0     0     0     0     0     0
     5    13     1     1     0     0     0     0     0     0     0
     6     1     1     1     0     0     0     0     0     0     0
     7     0     0     0     0     0     0     0     0     0     0
     8     3     1     1     0     0     0     0     0     0     0
     9     0     0     0     0     0     0     0     0     0     0
    10     0     0     0     0     0     0     0     0     0     0
    11     0     0     0     0     0     0     0     0     0     0
    12     0     0     0     0     0     0     0     0     0     0
    13     0     0     0     0     0     0     0     0     0     0
    14     0     0     0     0     0     0     0     0     0     0
    15     0     0     0     0     0     0     0     0     0     0
    16     0     0     0     0     0     0     0     0     0     0
    17     0     0     0     0     0     0     0     0     0     0
    18     0     0     0     0     0     0     0     0     0     0
    19     0     0     0     0     0     0     0     0     0     0
    20     0     0     0     0     0     0     0     0     0     0
    21     0     0     0     0     0     0     0     0     0     0
    22     0     0     0     0     0     0     0     0     0     0
    23     0     0     0     0     0     0     0     0     0     0
    24     0     0     0     0     0     0     0     0     0     0
    25     0     0     0     0     0     0     0     0     0     0
    26     0     0     0     0     0     0     0     0     0     0
    27     3     3     5     1     6     1     8     1     0     0
    28     0     0     0     0     0     0     0     0     0     0
    29     0     0     0     0     0     0     0     0     0     0
    30     0     0     0     0     0     0     0     0     0     0
    31     0     0     0     0     0     0     0     0     0     0
    32     0     0     0     0     0     0     0     0     0     0
    33     0     0     0     0     0     0     0     0     0     0
    34     7     2     3     1    27     1     0     0     0     0
    35     1     1    27     1     0     0     0     0     0     0
    36     1     1    27     1     0     0     0     0     0     0
    37     0     0     0     0     0     0     0     0     0     0
    38     0     0     0     0     0     0     0     0     0     0
    39     0     0     0     0     0     0     0     0     0     0
    40     0     0     0     0     0     0     0     0     0     0
    41     0     0     0     0     0     0     0     0     0     0
    42     0     0     0     0     0     0     0     0     0     0
    43     0     4     3     0    34     0    35     0    36     0
```

図 5.38: 再ブロック化法で実行して得られたタスクグラフ

図 5.37 により再ブロック化法で実行したことで 4 つブロックを発見し，それらのブロックが一気に新たなノードに変更した．変更したタスクグラフの行列は図 5.38 に示した．そしてこのタスクグラフのスケジュール結果は図 5.36 の右のように示した．

第 6 章

おわりに

6.1　研究の成果

本論文では，タスクグラフの実行時間を短縮するために，タスクグラフの構造に基づいてブロック化法とタスクグラフのスケジュール結果に基づいて再ブロック化法を提案した．

提案したブロック化法と再ブロック化法については，以下のような条件を前提としている．(1) ある一つのブロックに含まれる全てのノードは，同一のプロセッサで実行される．(2) ある一つのブロックに含まれるタスクを実行する場合，実行に必要な全てのデータは実行前に読み込まれ，全ての結果のデータは実行後に書き出される．

ブロック化法は，基本ブロック化法と追加ブロック化法からなっている．そして基本ブロック化法を先に適用し，次に追加ブロック化法を適用するように実行している．基本ブロック化法はタスクグラフの構造に基づいて提案した方法であり，４種類の親子ノードを対象に操作する．(1) 単純な親子ノード集合 (極小親子集合)，(2) 共通の先祖を持つ親子ノード集合 (同先祖親子集合)，(3) 共通の子孫を持つ親子ノード集合 (同子孫親子集合)，(4) 共通の先祖と子孫を持つ親子ノード集合 (同先祖子孫親子集合)．４種類の親子ノード集合のいずれかをもつ連結部分グラフを一つにすることは基本ブロック化法である．追加ブロック化法は各ノードの最早開始時刻と最遅開始時刻に基づいた提案した方法であり，４つの構造パターンを対象に操作する．(1) 多対一親子接続パターン，(2) 一対多親子接続パターン，(3) ある親ノードが別

の子ノードを持つ多対多親子接続パターン，(4) ある子ノードが別の親ノードを持つ多対多親子接続パターン．4 つの構造パターンのいずれかをもつ連結部分グラフを一つにすることは追加ブロック化法である．再ブロック化法はタスクグラフのスケジュール結果がさらに短縮するために，ブロック化法で実行されたタスクグラフにもう一度ブロック化する方法である．この方法は，スケジューリングの結果に基づいて行う．まず，ノードの実行に関する最早と最遅の開始時刻を用いた LST-EST スケジューリング法を提案し，このスケジューリングの実行結果に基づいて，一つのプロセッサで実行可能な連結部分グラフを一つにする．

提案したブロック化法と再ブロック化法を評価するためのシミュレーション方法およびその実験結果について示した．実験用のタスクグラフは MATLAB を用いて作成した自動車制御用の Simulink モデルから抽出して利用している．この自動車制御用の Simulink モデルは Coolant Control サブシステムモデル，Engine Output Control サブシステムモデル，Vehicle Control サブシステムモデル，Fuel Behavior Control サブシステムモデル，Induction Control サブシステムモデル，5 つのサブシステムモデルを含む．さらに各サブシステム中でサブサブシステムが存在する場合もある．実験用のタスクグラフの総ノード数は 429 個である．実行用のプロセッサの数は 2 つとする．提案したブロック化法を利用してスケジュール結果が 51%に減少し，提案した再ブロック化法を利用してスケジュール結果がさらに 8.8%に減少した．提案したブロック化法と再ブロック化法を適用した結果，通信時間が 55.3% ((490-219)/490 × 100) を短縮されたことを確認した．したがって，提案手法が自動車制御用システムの実行に有効であることが明らかである．

6.2 本研究の展望

近年の世界エネルギー情勢の中で，地域的に一番影響が大きかったのは東アジアである．東アジア各国の急成長とそれに伴うエネルギー需要の拡大がその理由になっている [50, 51, 52]．2002 年東アジア，太平洋諸島等におけるエネルギー消費量は世界のエネルギー消費量の 25 %近くを占めており，EU と並ぶに至っている．今後東アジア諸国，地域におけるエネルギー消費量は EU を上回るまでになると予測されている．東アジア各国中でもっともエネルギーの消費量が高い国は中国であり [53, 54, 55]，世界のエネルギー消費量の 12 %近くを占めている [56](図 6.1)．このような膨大なエネルギーの消費について，解決しなければならないのはエネルギーの提供量の問題だけでなく，エネルギーの伝送問題でもある [58, 59].

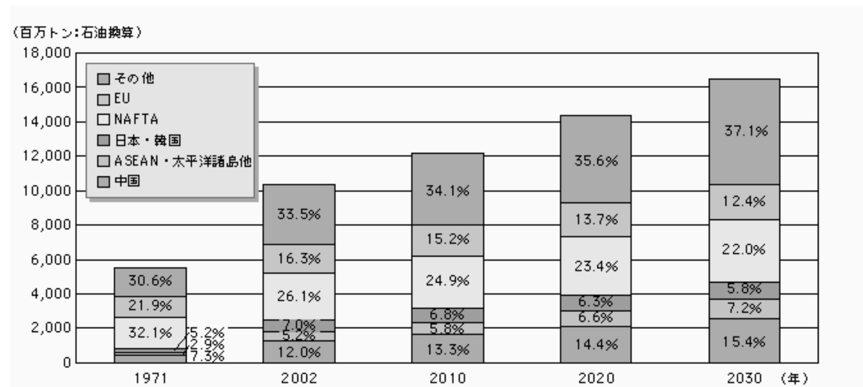


図 6.1: 世界の一次エネルギー消費量の推移 [56]

中国の場合，国内で生産されている電気は水力発電，原子力発電，火力発電などによる混在発電のものであるが [60, 61, 62]，鉱物であるエネルギー資源の分布が平均ではない (図 6.2) ため，鉱物不足の地域には発電所が発電できない問題がある [62]．このように，地域の電力が足りないことになっている．これを解決するために別の地域から長距離にわたっての電力の伝送が必要である．電力を伝送する時に発電所

から高い電圧を持つ電力が各変圧器を通過することによって電圧が徐々に低くなってくる．そして，受配電システムを通してそれぞれ相応の場所に電力を送っている．現在緊急時の電力を融通するため，各地域の受配電システムに送電網で繋ぎ，地域間連系線 (図 6.3) と呼ぶ．各地域の受配電システムが本地域の各変圧器を制御するだけにとどまらず，別の地域の各変圧器も制御することができる．

今受配電システムの多地域の制御方式に対して 2 つの方式が存在する．集中式制御と分散式制御である．集中式制御は別の地域から電力を受けて，自分の受配電システムを利用して各変圧器を制御する方式である．しかし，この制御方式はだんだん膨大な変圧器ネットワークが制御しにくくなる．分散式制御は各地域の複数受配電システムを利用して各変圧器を制御する方式である．しかし，この制御方式は電力が変圧器間に流れる時に異なる受配電システム間の情報交換の時間がかかるため，各場所に電力を送るまで相当の時間を要する，制御の性能が下がる．分層協調式は新しく提案した多地域の制御方式であり，集中式制御と分散式制御の特徴を持っている [64].

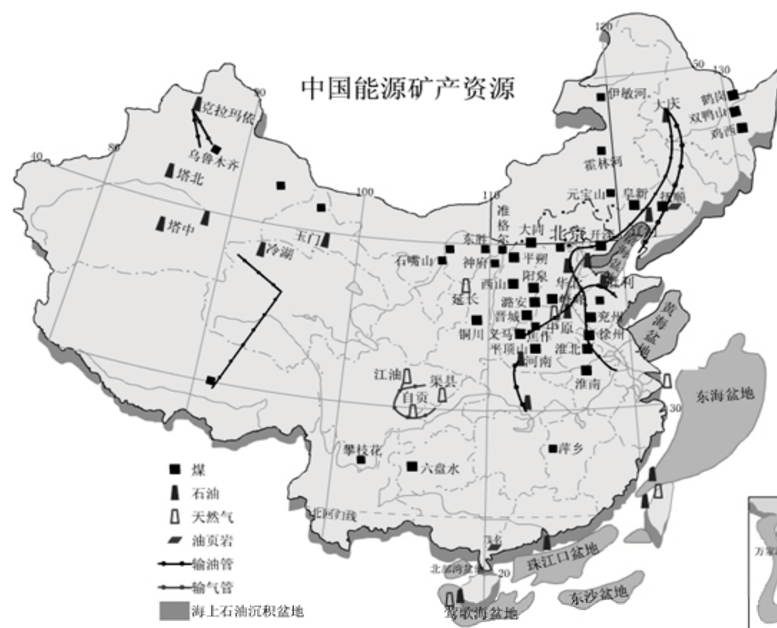


図 6.2: 中国の鉱物の分布 [57]

分散式制御の問題は，本論文で提案したブロック化の方法を利用して解決可能で

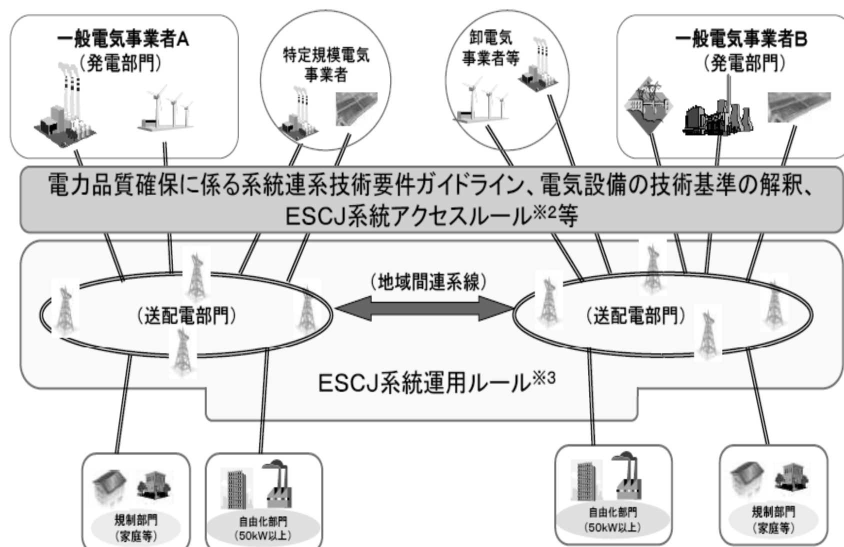


図 6.3: 地域間連系線 [63]

あると考えられる．それは，上流から下流へ繋がっている変圧器ネットワークをタスクグラフ，受配電システムにおける制御システムをプロセッサと見なし，そして変圧器ネットワークの部分ネットについて，ブロック化を行えば，効率よく電力の伝送ができるのではないかと思われる．したがって，本研究で得られる成果を，エネルギー伝送を含む社会システムにおける諸問題の解決に生かしていきたい．

6.3 今後の課題

本研究では，実用の自動車制御用システムに有効なブロック化手法を提案しているが，今後は適用対象のシステムを広げ，それぞれのシステムの特性や構造特徴等について本研究の提案手法を理論的かつ実験的に評価していく必要がある．特に，急成長が続いている東アジアや東南アジアの各国における社会システムの諸問題について，本研究成果を応用した解決法を検討していきたい．そのために，社会システムのグラフモデル作成やデータチューニングによるパラメータ設定，また本研究手法の改善をしていかなければならない．

参考文献

[1] “マルチプロセッサ・システム（multi-processor system）”, Insider’s Computer Dictionary, http://www.atmarkit.co.jp/icd/root/55/131189855.html, (参照 2015-01-12).

[2] “マルチプロセッサ”, Fundamental Information Techinology Engineer Examination, http://www.it-license.com, (参照 2015-01-12).

[3] Kasahara, H. and Narita, S.: “Parallel Processing for Real-time Control and Simulation of DCCS”, Proc. of 4th IFAC Workshop on Distributed Computer Control Systems, pp. 103-113 (1982).

[4] Ge, Q.W., Watanabe, T. and Onaga, K.: “Analysis of Parallelism in Autonomous Execution of Data-flow Program Nets”, IEICE Trans. Fundamentals, Vol. 74, No. 10, pp. 3008-3017 (1991).

[5] 川尻三重子, 葛崎偉, 中田充, 八木潔, 斗納 宏敏: “自動車制御システムにおけるマルチプロセッサスケジューリングについて”, 電子情報通信学会技術研究報告. CST, コンカレント工学, Vol. 102, No. 259, pp. 7-10, (2002).

[6] 斗納宏敏, 山田典生, 安田武史, 矢倉信次: “エンジン制御 ECU 用高機能ソフトチェッカ”, 富士通テン技報, Vol. 15, pp. 28-38 (1997).

[7] Weng, J.Z., Chen, X. and Lorenz T.: “A Multi-thread Parallel Computation Method for Dynamic Simulation of Molecular Weight Distribution of Multisite

Polymerization”, Computers & Chemical Engineering, Vol. 82, No. 2, pp. 55-67 (2015).

[8] Sreerama, K.R. and Chandrasekharan, E.: “A Parallel Distributed Computing Framework for Newton-Raphson Load Flow Analysis of Large Interconnected Power Systems”, International Journal of Electrical Power & Energy Systems, Vol. 73, pp. 1-6 (2015).

[9] Abu-Khzam, F.N., Daudjee, K., Mouawad, A.E. and Nishimura, N.: “On Scalable Parallel Recursive Backtracking”, Journal of Parallel and Distributed Computing, Vol. 84, pp. 65-75 (2015).

[10] Aristidou, P. and VanCutsem, T.: “A parallel Processing Approach to Dynamic Simulations of Combined Transmission and Distribution Systems”, International Journal of Electrical Power & Energy Systems, Vol. 72, pp. 58-65 (2015).

[11] Songkram, N. and Puthaseranee, B.: “E-learning System in Virtual Learning Environment to Enhance Cognitive Skills for Learners in Higher Education”, Procedia-Social and Behavioral Sciences, Vol. 174, No. 12, pp. 776-782 (2015).

[12] Schroer, A. and Handel, R.B.: “Data Bases and Statistical Systems: Education, General”, International Encyclopedia of the Social & Behavioral Sciences (Second Edition), pp. 763-768 (2015).

[13] Azeiteiro, U.M., Bacelar-Nicolau, P., Caetano, F.J.P. and Caeiro, S.: “Education for Sustainable Development Through E-learning in Higher Education: Experiences from Portugal”, Journal of Cleaner Production, Vol. 106, No. 1, pp. 308-319 (2015).

[14] 清水さや子, 横田賢史, 戸田勝善: “東京海洋大学における IC カード学生証の運用・評価および今後の展開”, 学術情報処理研究, No. 13, pp. 64-73 (2009).

[15] Kuramae, H., Shimano, A., Kimura, A., Matsumoto, M., Furuno, Y. and Kamejima, K.: "Integration of User Account Database on Multiple OS Environment", Proc. of IEEE Region 10 Conference on Computers, Communications, Control And Power Engineering, Vol. I, pp. 176-179 (2002).

[16] Hu, D.H., Sun, Z.L. and Li, H.Q.: "An Overview of Medical Informatics Education in China", International Journal of Medical Informatics, Vol. 82, No. 5, pp. 448-466 (2013).

[17] 立田ルミ："大学における一般情報教育の現状と今後の動向：情報処理学会一般情報教育委員会調査を基に", 情報学研究, Vol. 4, pp. 27-38 (2015).

[18] 藤田彬, 藤田央, 田村直良: "国語教育的評価項目を考慮した機械学習による日本語文章の自動評価と評価モデルの構築", 自然言語処理, Vol. 19, No. 4, pp. 281-301 (2012).

[19] Barros, C.A., Silveira, L.F.Q., Valderrama, C.A. and Xavier-de-Souza, S.: "Optimal Processor Dynamic-energy Reduction for Parallel Workloads on Heterogeneous Multi-core Architectures", Microprocessors and Microsystems, Vol. 39, No. 6, pp. 418-425 (2015).

[20] Vu, L. and Alaghband, G.: "Novel Parallel Method for Association Rule Mining on Multi-core Shared Memory Systems", Parallel Computing, Vol. 40, No. 10, pp. 768-785 (2014).

[21] Ge, Q.W. and Yoshioka, N.: "An Optimal Two-Processor Scheduling for a Class of Program Nets via Hybrid Priority List", Transactions of Information Processing Society of Japan, Vol. 40, No. 5, pp. 2064-2071 (1999).

[22] Ge, Q.W.: "PARAdeg-Processor Scheduling for Acyclic SWITCH-less Program Nets", J. Franklin Institute, Vol. 336, No. 7, pp. 1135-1153 (1999).

[23] Garey, M.R. and Johson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Comppleteness, W.H. Freeman and Company, (1979).

[24] Coffman, E.G.: Computer and Job-Shop Scheduling Theory, John Wiley, (1976).

[25] Lenstra, J.K. and Kan, A.H.G.R.: "Conplexity of Scheduling Under Precedence Constraints", Operations Research, Vol. 26, pp. 22-35 (1978).

[26] Hoogeveen, J.A., Lenstra, J.K. and Veltman, B.: "Preemptive Scheduling in a Two-stage Multiprocessor Flow Shop is NP-hard", European Journal of Operational Research, Vol. 89, Issue 1, pp. 172-175 (1996).

[27] Hu, T.: "Parallel Sequencing and Assemply Line Problems", Operations Research, Vol. 9, pp. 841-848 (1961).

[28] Coffman, E.G. and Graham, R.: "Optimal Scheduling for Two-processor Systems", Acta Information, Vol. 1, pp. 200-213 (1972).

[29] Tanaka, S., Kouno, T. and Moriyama, Y.: "Development of Control Software by All-software Simulation", Fujitsu Ten Technical Report, Vol. 24, pp. 28-33 (2006).

[30] 大塚隆史, 養畑裕紀, 葛崎偉, 中田充, 森山裕, 斗納宏敏: "通信時間を考慮したマルチプロセッサスケジューリング手法の提案", 電子情報通信学会技術研究報告. CST, コンカレント工学, Vol. 107, No. 472, pp. 23-28 (2008).

[31] Li, Z.H., Liu, Y.K. and Yang, G.Q.: "A New Probability Model for Insuring Critical Path Problem with Heuristic Algorithm", Neurocomputing, Vol. 148, pp. 129-135 (2015).

[32] 右田雅裕, 多田昭雄, 糸川剛, 中村良三: "PERT チャートにおけるクリティカルパスを求める並列アルゴリズム", 情報処理学会論文誌, Vol. 47, No. 7, pp. 2212-2223 (2006).

[33] Brest, J. and Zumer, V.: "A Performance Evaluation of List Scheduling Heuristics for Task Graphs without Communication Costs", Proceedings of the 2000 International Workshops on Parallel Processing, pp. 421-428 (2000).

[34] Tanaka, M. and Tatebe, O.: "Workflow Scheduling to Minimize Data Movement Using Multi-Constraint Graph Partitioning", IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (2012).

[35] Liou, J. and Palis, M.A.: "An Efficient Task Clustering Heuristic for Scheduling DAGs on Multiprocessors", Information, Vol. 1, pp. 152–156 (1996).

[36] Song, Y.L. and Yu, M.H.: "On Finding the Longest Antisymmetric Path in Directed Acyclic Graphs', Information Processing Letters, Vol. 115, Issue 2, pp. 377-381 (2015).

[37] Tholey, T.: "Linear Time Algorithms for Two Disjoint Paths Problems on Directed Acyclic Graphs", Theoretical Computer Science, Vol. 465, pp. 35-48 (2012).

[38] 矢田哲士, 石川幹人, 田中秀俊, 浅井潔: "隠れマルコフモデルと遺伝的アルゴリズムによるDNA配列のシグナルパターン抽出", 情報処理学会論文誌, Vol. 37, No. 6, pp. 1117-1129 (1996).

[39] 仙石正和, Masakazu, S.: "一般化シグナルフローグラフにおける保存量", 電子情報通信学会論文誌. A, 基礎・境界 J65-A(1), pp. 62-68 (1982).

[40] 草部博輝, 中森眞理雄: "資源制約と先行制約を拡張したプロジェクトスケジューリング問題とその下界値の計算方法 (研究速報)", 電子情報通信学会論文誌. A, 基礎・境界 J91-A(4), pp. 517-521 (2008).

[41] Skaggs, T.H.: "Assessment of Critical Path Analyses of the Relationship Between Permeability and Electrical Conductivity of Pore Networks", Advances in Water Resources, Vol. 34, Issue 10, pp. 1335-1342 (2011).

[42] Odaka, A. and Kai, M.: “Task Granularity Analysis for Task Scheduling Tasking Account of Communication Overhead”, The Journal of the Faculty of Science and Technology, Vol. 44, pp. 17-23 (2007).

[43] 東達軌, 武田正之: “属性付きグラフ書き換え系による計算モデルの表現とその応用”, 情報処理学会論文誌プログラミング（PRO), Vol. 2, No. 5, pp. 44-44 (2009).

[44] Moussourakis, J. and Haksever, C.: “Models for Accurate Computation of Earliest and Latest Start Times and Optimal”, Compression in Project Networks, J. Constr. Eng. Manage, Vol. 133, pp. 600-608 (2007).

[45] Abraham, G.T., James, A. and Yaacob, N.: “Priority-grouping Method for Parallel Multi-scheduling in Grid”, Journal of Computer and System Sciences, Vol. 81, Issue 6, pp. 943-957 (2015).

[46] 田中清史: “適応型スケジューリングによる平均応答時間の短縮法—実行時間見積り方法の影響”, 情報処理学会論文誌, Vol. 55, No. 8, pp. 1856-1865 (2014).

[47] Santos, L.M.R., Munari, P., Costa, A.M. and Santos, R.H.S.: “A Branch-price-and-cut Method for the Vegetable Crop Rotation Scheduling Problem with Minimal Plot Sizes”, European Journal of Operational Research, Vol. 245, Issue 2, pp. 581-590 (2015).

[48] Tarjan, R. E.: “Depth-first search and linear graph algorithms”, SIAM J. Computing 1, pp. 146-160 (1972).

[49] Stuart Russell and Peter Norvig: “Artificial Intelligence: A Modern Approach (2nd ed.)”, Upper Saddle River, NJ: Prentice Hall, ISBN 0-13-790395-2 (2003).

[50] 恒川潤：“東アジアのエネルギー需給と戦略環境への影響”，防衛研究所紀要, Vol. 4, No. 2, pp. 66-88 (2001).

[51] 張文青: “エネルギー・環境分野をめぐる域内協力–東アジア経済共同体の結成に向けて”, 立命館国際地域研究, No. 22 (2004).

[52] 唐彦林: “国際政治背景下的東アジア能源戦略問題”, 中国社会科学网, Vol. 11 (2005).

[53] 伊藤庄一: “中国のエネルギー需要急増と日中関係–北東アジア・エネルギーダイナミズム再考”, Erina report, Vol. 85, pp. 36-46 (2009).

[54] “膨張続ける中国経済と東アジアエネルギー情勢”, 旬刊セキツウ (2188), pp. 33-37 (2004).

[55] 李志東, 伊藤浩吉, 小宮山涼一, “中国 2030 年エネルギー需給展望と北東アジアエネルギー共同体の検討”, エネルギーシステム・経済・環境コンファレンス講演論文集, Vol. 21, pp. 409-412 (2005).

[56] “第 I 部　東アジアとの新たな関係と国土交通施策の展開”, 国土交通省, http://www.mlit.go.jp/hakusyo/mlit/h16/hakusho/h17/html/g1023100.html, (参照 2015-05-18).

[57] “中国能源鉱産資源”, 中国地理, http://www.eku.cc/xzy/sctx/121255.htm, (参照 2015-07-11).

[58] 劉景林: “能源与運輸”, 学習与探索, Vol. 2, pp. 76-83 (1985).

[59] 林伯強, 姚　: “電力布局優化与能源総合運輸体系”, 経済研究, Vol. 6, pp.105-115 (2009).

[60] “水力中国, 中国三峡水力ダム発電所の現状について”, 海外電力, Vol. 55, No. 8, pp. 59-61 (2013).

[61] 渡辺搖: “原子力発電所建設時代を迎えた中国–原子力発電の現状と方向性”, 日中経協ジャーナル, No. 194, pp. 16-21 (2010).

[62] 孫亮, 黄偉隆, 馬瑞: “実施 CCS 技術的火力発電所の場所の選択”, 生態経済, Vol. 2, pp. 76-80 (2013).

[63] “送配電システムの現状と課題について次世代送配電ネットワーク研究会の概要等”, 資源エネルギー庁電力ガス事業部, http://www.meti.go.jp/committee/materials2/downloadfiles/g100527a05j.pdf, (参照 2015-07-11).

[64] 王成山, 王丹, 周越: “智能配電システム架構分析及技術挑戦”, 電力システム自動化, Vol. 39, No. 9 (2015).

付録 I: MATLAB/Simulinkでモデル化される車両制御システム

ここでは実験用システムモデルを紹介する．図 6.4 は MATLAB/Simulink でモデル化される車両制御システムである．シミュレション結果と考察についてこの車両制御システムモデルのタスクグラフを利用して実験を行う．この車両制御システムモデルは Coolant Control サブシステムモデル (図 6.5)，Engine Output Control サブシステムモデル (図 6.7)，Vehicle Control サブシステムモデル (図 6.14)，Fuel Behavior Control サブシステムモデル (図 6.15)，Induction Control サブシステムモデル (図 6.21)，5 つのサブシステムモデルを含む．さらに各サブシステム中でサブシステムが存在する場合もある．次は図 6.4 に含まれる各サブシステムの構造を詳しい表示する．

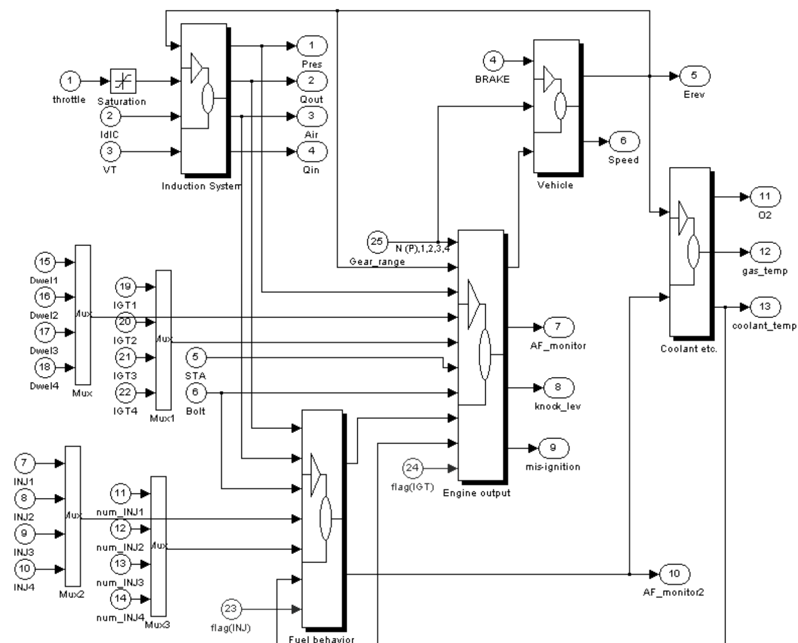


図 6.4: 車両制御システムモデル model1

Coolant Control サブシステムモデル (図 6.5) は 1 つのサブシステム (図 6.6) を含む.

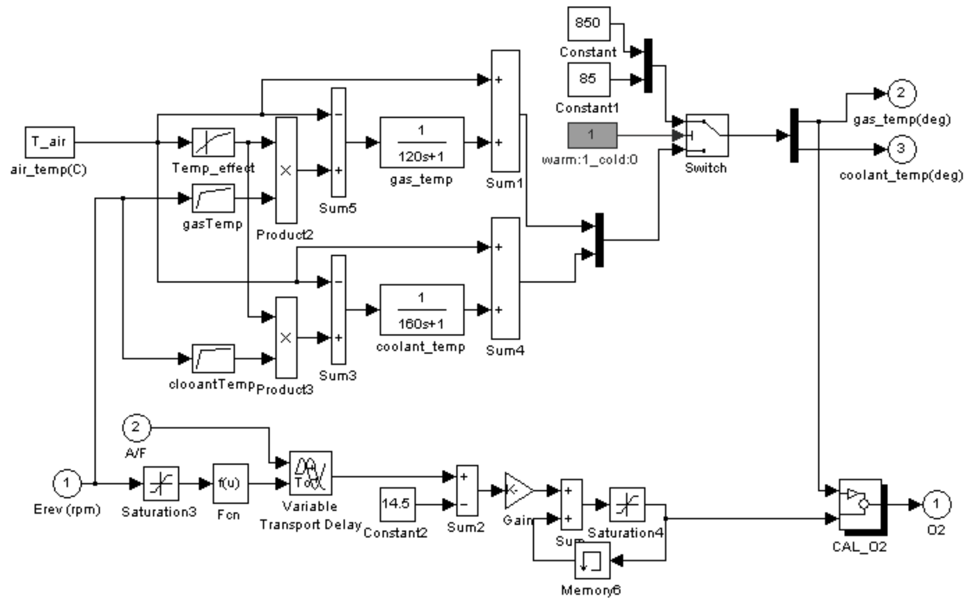


図 6.5: Coolant Control サブシステムモデル model1/Coolant etc

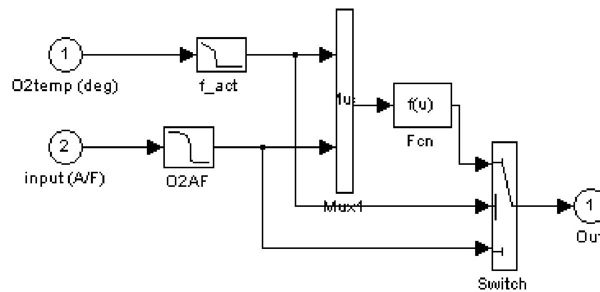


図 6.6: CAL_O2 サブシステムモデル model1/Coolant etc/CAL_O2

Engine Output Control サブシステムモデル (図 6.7) は 6 つのサブシステム (図 6.8, 6.9, 6.10, 6.11, 6.12, 6.13) を含み，中には図 6.9 がサブサブシステムである．

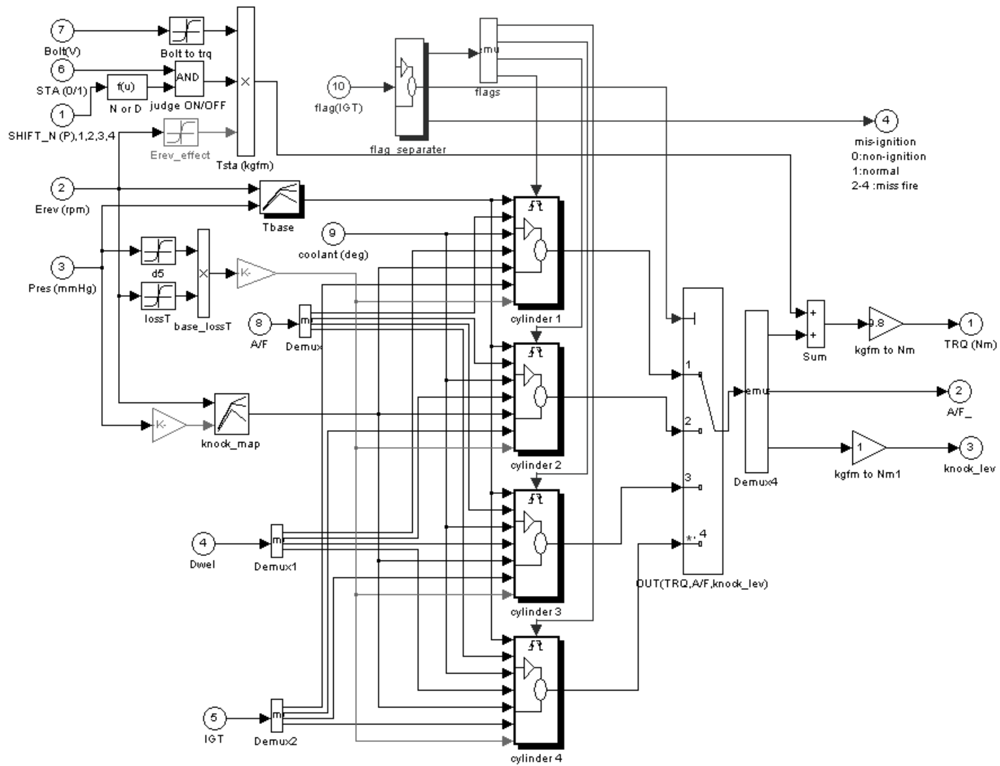


図 6.7: Engine Output Control サブシステムモデル model1/Engine output

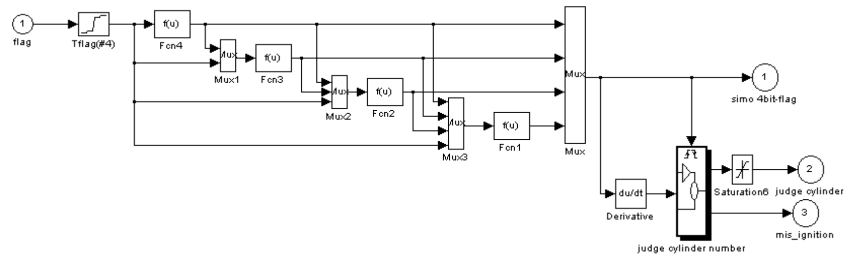


図 6.8: flag_separater サブシステムモデル model1/Engine output/flag_separater

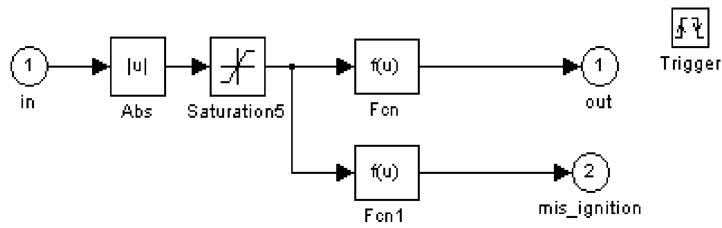


図 6.9: judge cylinder number サブシステムモデル model1/Engine output/
flag_separater/judge cylinder number

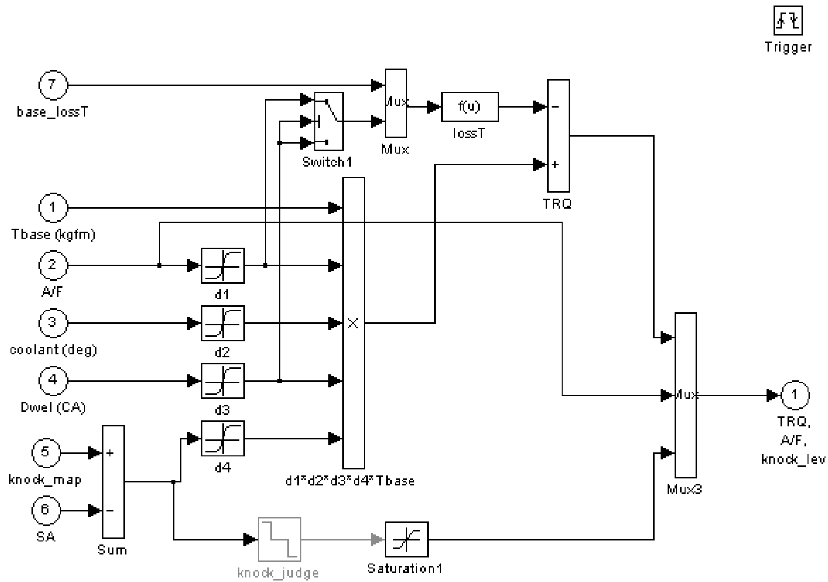


図 6.10: cylinder 1 サブシステムモデル model1/Engine output/cylinder 1

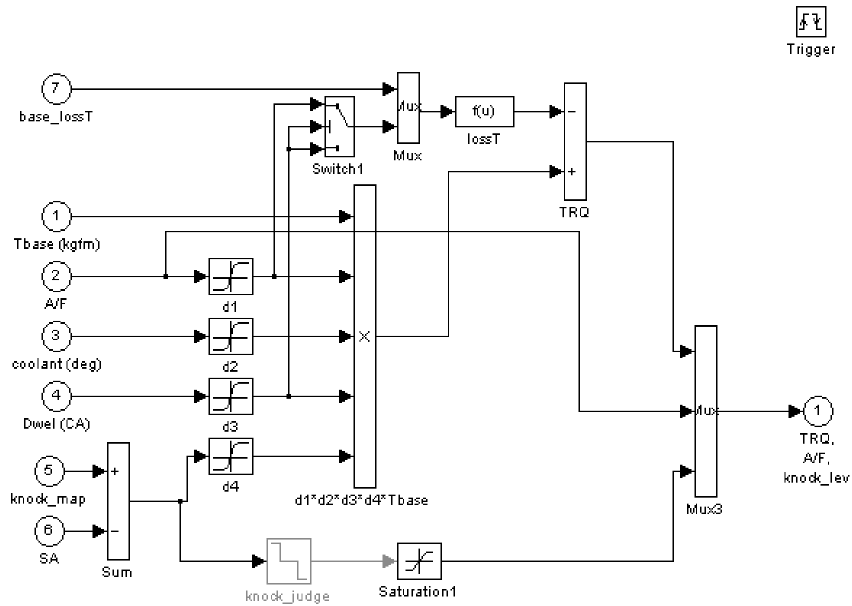


図 6.11: cylinder 2 サブシステムモデル model1/Engine output/cylinder 2

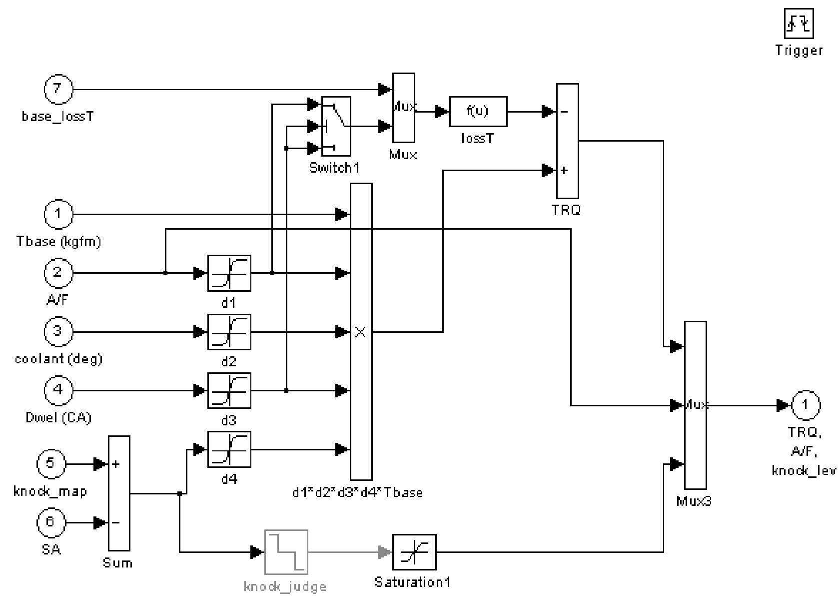


図 6.12: cylinder 3 サブシステムモデル model1/Engine output/cylinder 3

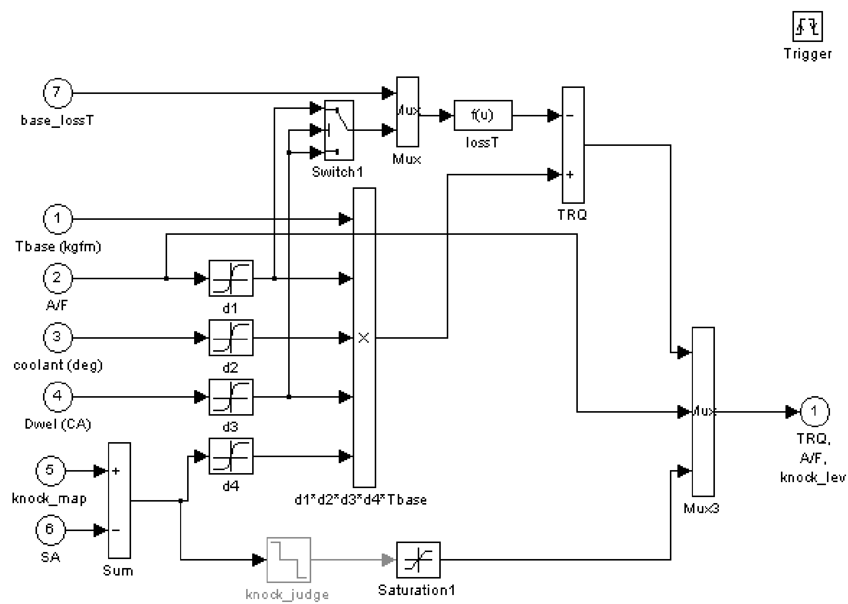


図 6.13: cylinder 4 サブシステムモデル model1/Engine output/cylinder 4

Vehicle Control サブシステムモデル (図 6.14) のサブシステムが存在しない.

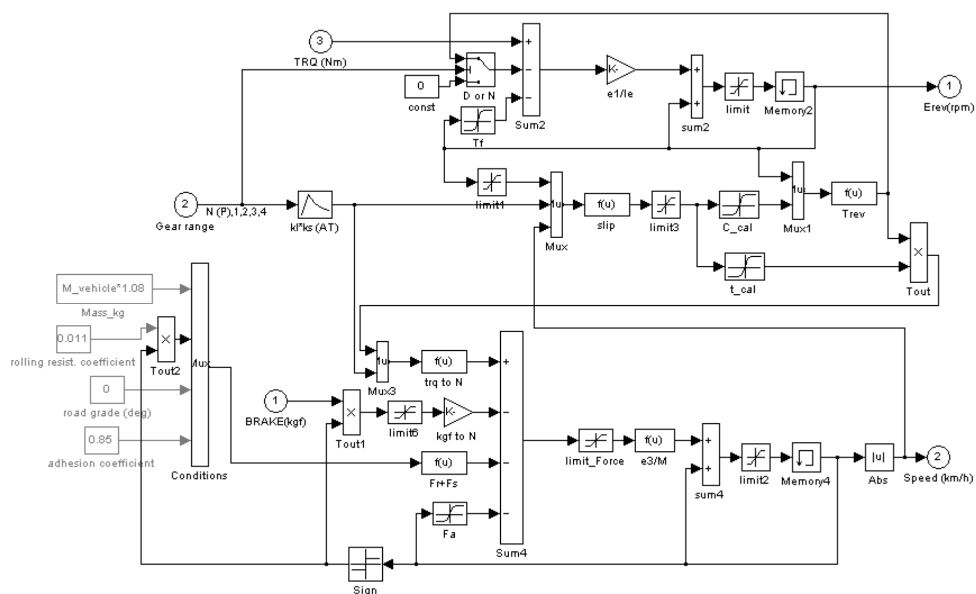


図 6.14: Vehicle Control サブシステムモデル model1/Vehicle

Fuel Behavior Control サブシステムモデル (図 6.15) は 5 つのサブシステム (図 6.16, 6.17, 6.18, 6.19, 6.20) を含む.

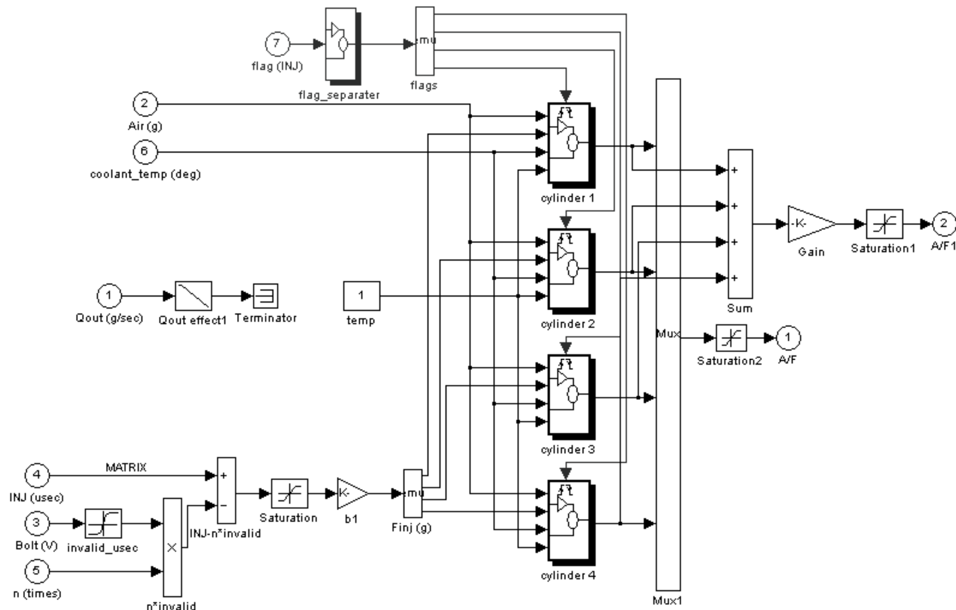


図 6.15: Fuel Behavior Control サブシステムモデル model1/Fuel behavior

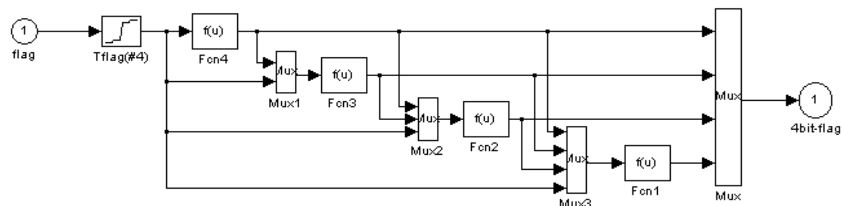


図 6.16: flag_separater サブシステムモデル model1/Fuel behavior/flag_separater

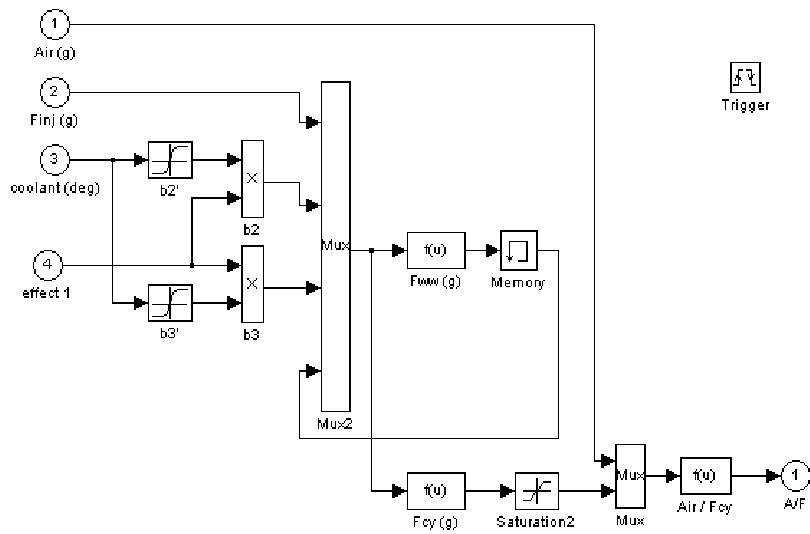


図 6.17: cylinder 1 サブシステムモデル model1/Fuel behavior/cylinder 1

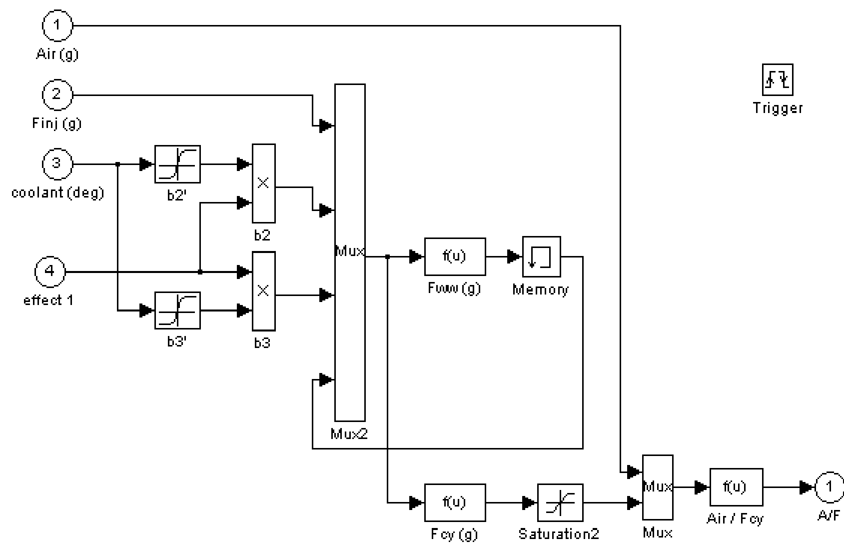


図 6.18: cylinder 2 サブシステムモデル model1/Fuel behavior/cylinder 2

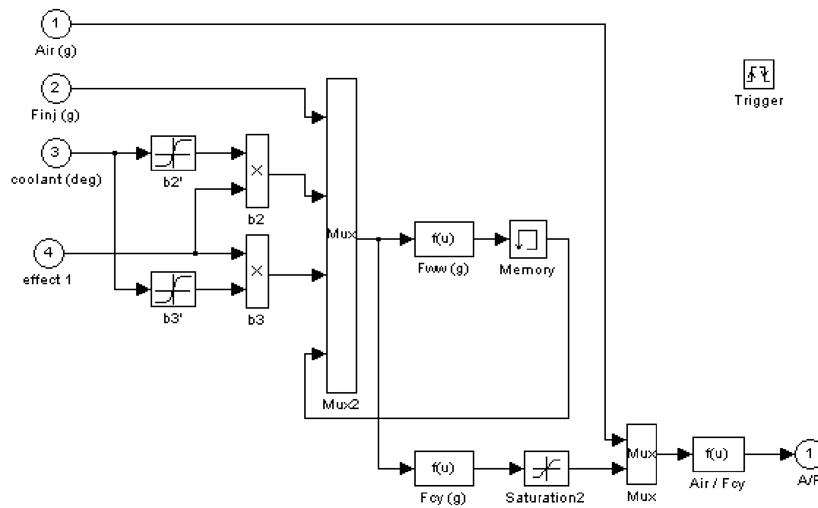


図 6.19: cylinder 3 サブシステムモデル model1/Fuel behavior/cylinder 3

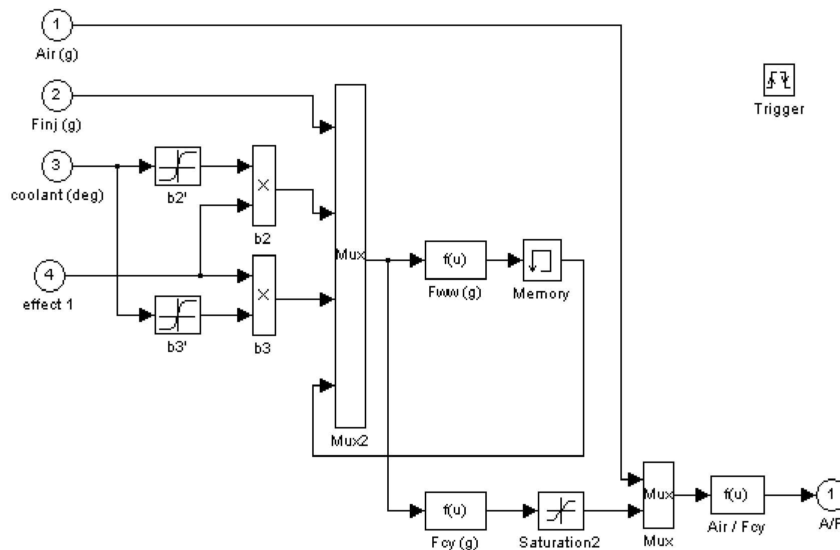


図 6.20: cylinder 4 サブシステムモデル model1/Fuel behavior/cylinder 4

Induction Control サブシステムモデル (図 6.21) は 1 つのサブシステム (図 6.22) を含む.

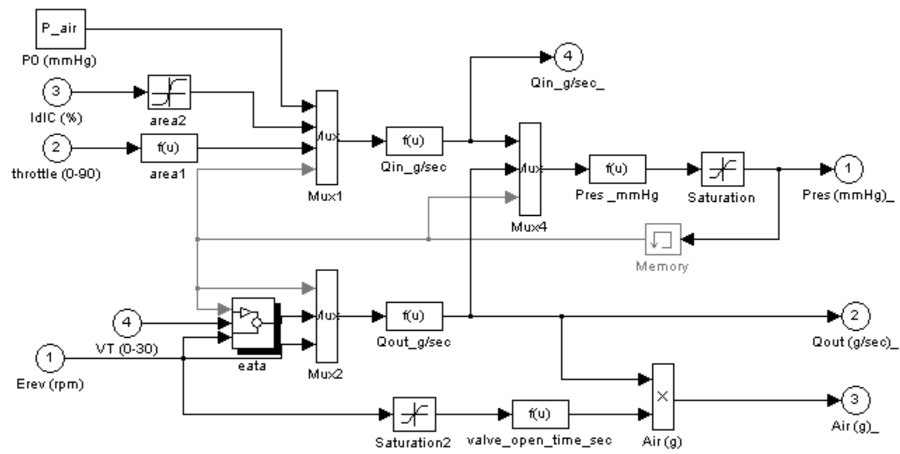


図 6.21: Induction Control サブシステムモデル model1/Induction System

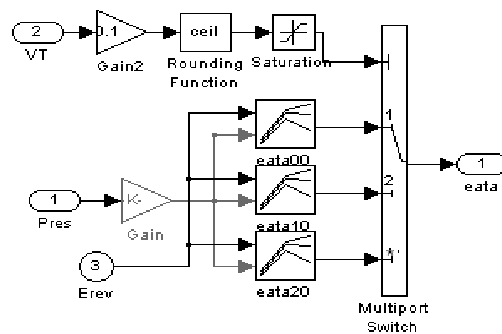


図 6.22: eata サブシステムモデル model1/Induction System/eata

付録 II: Matlab で作成されたスクリプト M-ファイル (実行用プログラム)

（１）まず Simulink モデルの隣接情報抽出のプログラムを示す．本プログラムの実行ファイルは abstract.m という名前のスクリプト M-ファイルである．

```
 1     %キーボードからモデル名を得て変数に格納
 2 -   modelName = input('モデル名を入力してください：','s');
 3     %モデルを開く
 4 -   open_system(modelName)
 5     %モデル内全てのブロック名を取得
 6 -   blkName = find_system(modelName, 'Type', 'block');
 7     %モデル内全てのブロックのハンドルを取得
 8 -   hBlkName = find_system(modelName,'FindAll','On','type', 'block');
 9     %ブロック名に番号を昇順に振る
10 -   blkNo = hBlkName;
11     %モデル内のブロックの数をカウント
12 -   countBlk = length(blkName);
13
14
15     %１つのブロックに対する親の数をカウント
16 -   for i=1:countBlk
17 -       hLine = get_param(blkNo(i),'LineHandles'); %ラインのハンドル取得
18 -       tmp(i) = length(hLine.Inport); %ブロックの入力数をカウント
19 -   end
20
21
22     %task初期化
23 -   row = max(tmp); %親の数の最大数を計算
24 -   task = zeros(countBlk+3,2*row+3); %全ての要素がゼロの行列作成
25
26     %ノード数を入力
27 -   task(1,1)=countBlk;
28     %ソースノードの情報
29 -   task(2,1)=1; %ソースノード番号
30 -   task(2,2)=0; %ソースノードの処理時間
31 -   task(2,3)=0; %ソースノードの実行時間
32
33     %シンクノード情報
34 -   task(countBlk+3,1)=countBlk+2; %シンクノード番号
35 -   task(countBlk+3,2)=0; %シンクノード実行時間
36
37     %隣接関係を調べてtaskデータを作成
38 -   for i=1:countBlk;
39         %ブロックのタイプを調べる
40 -       if strcmp(get_param(blkName(i), 'BlockType'), 'SubSystem')
```

図 6.23: abstract.m（１）

```
41          % 隣接関係を調べる対象がサブシステムなら以下を実行する
42 -        task(i+2,1)=i+1; %SubSystemであればノード番号のみ追加
43
44 -        hLine = get_param(blkNo(i),'LineHandles'); %ラインのハンドル取得
45 -        InportLine = hLine.Inport;
46
47 -        for j = 1 : length(InportLine)
48              %
49 -            DstPort_handle = get_param(InportLine(j), 'DstPortHandle');
50 -            DstPort_handle = DstPort_handle(DstPort_handle ~= -1 );     % 無効ポート(-1)を除外する
51
52
53 -            for k = 1 : length(DstPort_handle)
54                  %ブロックを取得する
55                  % ポートのあるブロック (あるいはサブシステム)
56 -                DstPort_parent = get_param(DstPort_handle(k), 'Parent');
57                  DstPortName = find_system(DstPort_parent,'SearchDepth','1','Port',int2str(get_param(DstPort_handle(k), 'PortNumber')),
58                  'BlockType', 'Inport') ;  %PortNumberに対応するInport端子の名前取得
59
60                  %親ノードの探索
61 -                SrcPort_handle=get_param(InportLine(j),'SrcPortHandle');
62 -                SrcPort_parent=get_param(SrcPort_handle,'Parent');
63
64 -                if strcmp(get_param(SrcPort_parent, 'Type'), 'block') % 親がブロックの場合
65
66                      % 親がサブシステムの場合
67 -                    if strcmp(get_param(SrcPort_parent, 'BlockType'), 'SubSystem')
68                          SrcPortName = find_system(SrcPort_parent, 'SearchDepth','1','Port', int2str(get_param(SrcPort_handle,
69                          'PortNumber')), 'BlockType', 'Outport');  %PortNumberに対応するOutport端子の名前取得
70
71                          %親のノード番号探索
72 -                        for y=1:countBlk
73 -                            if strcmp(SrcPortName,blkName(y))
74                                  %taskにデータ追加
75 -                                for z=1:countBlk
76 -                                    if strcmp(DstPortName,blkName(z))
77 -                                        task(z+2,1)=z+1;
78 -                                        task(z+2,2)=0;  %実行時間 %Inportブロックの実行時間は0
79 -                                        task(z+2,3)=1;  %親の数
80 -                                        task(z+2,4)=y+1;    %親のノード番号
81 -                                        task(z+2,5)=1;      %通信時間
82 -                                    end
83 -                                end
84 -                            end
85 -                        end
86 -                    else
87
88                          %親がサブシステム以外の時
89 -                        for y=1:countBlk
90                              %親のノード番号探索
```

図 6.24: abstract.m（２）

```
 91 -                 if (get_param(InportLine(j),'SrcBlockHandle')==blkNo(y))
 92                       %taskに追加
 93 -                     for z=1:countBlk
 94 -                         if strcmp(DstPortName,blkName(z))
 95 -                             task(z+2,1)=z+1;
 96 -                             task(z+2,2)=0;  %実行時間 %Inportブロックの実行時間は0
 97 -                             task(z+2,3)=1;  %親の数
 98 -                             task(z+2,4)=y+1;    %親のノード番号
 99 -                             task(z+2,5)=1;      %通信時間
100 -                         end
101 -                     end
102 -                 end
103 -             end
104 -         end
105 -     end
106 -     end
107 - end
108 - else
109       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
110       %現在検索対象のブロックがサブシステム以外で、
111       %taskに値が挿入されていない時に以下を実行する
112       %これは現在の検索対象がサブシステムであった時既にInportブロックに対して隣接関係を抽出しており、重複を避けるため
113 -     if task(i+2,:)==0
114           %if ~strcmp(get_param(blkName(i), 'BlockType'), 'Inport')
115           %ブロックに接続しているラインのハンドルを取得
116 -         hLine = get_param(blkNo(i),'LineHandles');
117           %ブロックに入力されているライン(親)の数をカウント
118 -         countLine = length(hLine.Inport);
119
120 -         if strcmp(get_param(blkName(i), 'BlockType'), 'Outport')
121 -             task(i+2,1)=i+1;    %taskの指定の位置にノード番号格納
122 -             task(i+2,2)=0;    %処理時間格納 %Outportの実行時間は0
123 -             task(i+2,3)=countLine;  %親の数格納
124
125               %親が存在する場合のノードの処理
126 -             if countLine~=0
127                   %現在注目しているブロックが単純ブロックかOutportブロックかで場合分け
128                   %各ラインの入力先のブロックハンドルを取得
129 -                 for n=1:countLine
130 -                     hSrcBlk = get_param(hLine.Inport(n),'SrcBlockHandle');
131
132                       %親のブロックタイプ判定
133 -                     if strcmp(get_param(hSrcBlk, 'BlockType'), 'SubSystem')
134                           %親がサブシステムの時
135                           %親ノードの探索
136 -                         SrcPort_handle=get_param(hLine.Inport(n),'SrcPortHandle');
137 -                         SrcPort_parent=get_param(SrcPort_handle,'Parent');
138                           SrcPortName = find_system(SrcPort_parent,'SearchDepth','1','Port',
139                           int2str(get_param(SrcPort_handle, 'PortNumber')), 'BlockType', 'Outport');
140                           %PortNumberに対応するOutport端子の名前取得
```

図 6.25: abstract.m（3）

```
141
142                         %親のノード番号探索
143 -                       for y=1:countBlk
144 -                           if strcmp(SrcPortName,blkName(y))
145                                 %taskにデータ追加
146 -                               task(i+2,2*n+2)=y+1;    %親のノード番号
147 -                               task(i+2,2*n+3)=1;      %通信時間
148
149 -                           end
150 -                       end
151 -                   else
152                         %親がサブシステム以外の時
153                         %入力先ブロックハンドルと一致するブロック番号を格納
154 -                       for j=1:countBlk
155 -                           k = (blkNo(j)==hSrcBlk);
156 -                           if k==1
157 -                               task(i+2,2*n+2)=j+1;   %親ノード
158 -                               task(i+2,2*n+3)=0;   %通信時間
159 -                           end
160 -                       end
161 -                   end
162 -               end
163 -           end
164 -       else
165 -           if strcmp(get_param(blkName(i), 'BlockType'), 'Inport')
166 -               task(i+2,1)=i+1;    %taskの指定の位置にノード番号格納
167 -               task(i+2,2)=0;    %処理時間格納
168 -               task(i+2,3)=1;  %親の数格納
169 -           else
170 -               task(i+2,1)=i+1;    %taskの指定の位置にノード番号格納
171 -               task(i+2,2)=1;    %処理時間格納
172 -               task(i+2,3)=countLine;  %親の数格納
173 -           end
174             %親が存在する場合のノードの処理
175 -           if countLine~=0
176                 %現在注目しているブロックが単純ブロックかOutportブロックかで場合分け
177
178                 %各ラインの入力先のブロックハンドルを取得
179 -               for n=1:countLine
180 -                   hSrcBlk = get_param(hLine.Inport(n),'SrcBlockHandle');
181
182                     %親のブロックタイプ判定
183 -                   if strcmp(get_param(hSrcBlk, 'BlockType'), 'SubSystem')
184                         %親がサブシステムの時
185                         %親ノードの探索
186 -                       SrcPort_handle=get_param(hLine.Inport(n),'SrcPortHandle');
187 -                       SrcPort_parent=get_param(SrcPort_handle,'Parent');
188                         SrcPortName = find_system(SrcPort_parent,'SearchDepth','1','Port',
189                         int2str(get_param(SrcPort_handle, 'PortNumber')), 'BlockType', 'Outport');
190                         %PortNumberに対応するOutport端子の名前取得
```

図 6.26: abstract.m（4）

```
191
192                             %親のノード番号探索
193 -                           for y=1:countBlk
194 -                               if strcmp(SrcPortName,blkName(y))
195                                     %taskにデータ追加
196 -                                   task(i+2,2*n+2)=y+1;    %親のノード番号
197 -                                   task(i+2,2*n+3)=1;      %通信時間
198
199 -                               end
200 -                           end
201 -                       else
202                             %親がサブシステム以外の時
203                             %入力先ブロックハンドルと一致するブロック番号を格納
204 -                           for j=1:countBlk
205 -                               k = (blkNo(j)==hSrcBlk);
206 -                               if k==1
207 -                                   if ~strcmp(get_param(blkName(j), 'BlockType'), 'Inport')
208 -                                       task(i+2,2*n+2)=j+1;   %親ノード
209 -                                       task(i+2,2*n+3)=1;   %通信時間
210 -                                   else
211 -                                       task(i+2,2*n+2)=j+1;   %親ノード
212 -                                       task(i+2,2*n+3)=0;   %通信時間
213                                         %%disp(j);
214 -                                   end
215 -                               end
216 -                           end
217 -                       end
218 -                   end
219 -               end
220 -           end
221 -       end
222 -   end
223 - end
224
225   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
226   %ソースノードとシンクノードの処理
227 - p=0;    %p初期化(pは子が存在しないノードの数)
228   %最上位層モデル内全てのブロックのハンドルを取得
229 - hTopBlkName = find_system(modelName,'SearchDepth','1','FindAll','On','type','block');
230   %最上位層モデル内のブロックの数をカウント
231 - TopCountBlk = length(hTopBlkName);
232
233 - for i=1:TopCountBlk
234 -     hTopLine = get_param(hTopBlkName(i),'LineHandles');
235       %入力がなければソースノードと連結
236 -     if isempty(hTopLine.Inport)
237 -         for j=1:countBlk
238 -             if hBlkName(j)==hTopBlkName(i)
239 -                 task(j+2,4)=1; %ソースノードと連結
240 -                 task(j+2,5)=0; %ソースノードとの通信時間
```

図 6.27: abstract.m（５）

```
241 -                 end
242 -             end
243 -         end
244           %出力がなければシンクと連結
245 -         if isempty(hTopLine.Outport)
246 -             p=p+1;  %シンクノードの親の数をカウント
247 -             for k=1:countBlk
248 -                 if hBlkName(k)==hTopBlkName(i)
249 -                     task(countBlk+3,2*p+2)=k+1; %シンクノードとの接続情報を追加
250 -                     task(countBlk+3,2*p+3)=0; %シンクノードとの通信時間
251 -                 end
252 -             end
253 -         end
254 -     end
255
256       %シンクノードの親のノード数を追加
257 -     task(countBlk+3,3)=p;
258
259       %disp(task(countBlk+3,:));
260
261       %disp(task);
262
263 -     for i=1:countBlk+2;
264
265 -         Point(i,1)=i;
266 -         Point(i,2)=0;
267 -         Point(i,3)=0;
268
269 -         Check(i)=0; %探索チェック
270 -         parCheck(i)=0; %子の個数
271 -         par(i,1)=i; %子の格納
272 -         CurHldB(i)=0; %探索済みのi
273 -         CurHldC(i)=0; %探索済みのj
274 -         LoopCheck(i)=0; %ループチェック
275 -         if task(i,2)~=0 && task(i,3)==0
276 -            task(i,3)=task(i,3)+1;
277 -            task(i,4)=1;
278 -            task(i,5)=0;
279 -         end
280
281 -     end
282       %%disp(task);
283
284
285 -     Check(1)=1;
286 -     Check(countBlk+2)=-1;
287 -     Cur=1;
288 -     jack=0;
289 -     CurHldA(1)=0;
290
```

図 6.28: abstract.m（6）

```
291 -  while Check(1)~=-1
292
293 -      if CurHldB(Cur)~=0
294 -          Au=CurHldB(Cur);
295 -          CurHldB(Cur)=0;
296 -      else
297 -          Au=1;
298 -      end
299 -      for i=Au: countBlk+2
300 -          if CurHldC(Cur)~=0
301 -              Bu=CurHldC(Cur)+1;
302 -              CurHldC(Cur)=0;
303 -          else
304 -              Bu=1;
305 -          end
306
307 -          for j=Bu: task(i+1,3)
308 -              k=2*(j+1);
309
310 -              if task(i+1,k)==Cur
311                    %多重辺の探索%
312 -                  if parCheck(Cur)>0
313 -                      for m=1: parCheck(Cur)
314 -                          if par(Cur,m+2)==i
315 -                              jack=1;
316 -                          end
317 -                      end
318 -                  end
319
320                    %子の個数を追加%
321 -                  if jack==0
322 -                      parCheck(Cur)=parCheck(Cur)+1;
323 -                      par(Cur,parCheck(Cur)+2)=i;
324 -                      par(Cur,2)=parCheck(Cur);
325 -                  end
326
327                    %ループまたは多重辺の処理%
328 -                  if Check(i)==1 || jack==1
329 -                      task(i+1,k)=0;
330 -                      task(i+1,k+1)=0;
331 -                      task(i+1,3)=task(i+1,3)-1;
332 -                      for m=j: task(i+1,3)
333 -                          k=2*(m+1);
334 -                          task(i+1,k)=task(i+1,k+2);
335 -                          task(i+1,k+2)=0;
336 -                          task(i+1,k+1)=task(i+1,k+3);
337 -                          task(i+1,k+3)=0;
338 -                      end
339 -                      jack=0;
340                        %ループチェック%
```

図 6.29: abstract.m（7）

```
341 -                 if Check(i)==1
342 -                   LoopCheck(Cur)=1;
343 -                   par(Cur,parCheck(Cur)+2)=0;
344 -                   parCheck(Cur)=parCheck(Cur)-1;
345 -                   par(Cur,2)=parCheck(Cur);
346 -                 end
347 -                 CurHldB(Cur)=i;
348 -                 CurHldC(Cur)=j-1;
349 -                 i=countBlk+3;
350 -                 break;
351 -               end
352
353               %次の訪問先へ%
354 -             if Check(i)==0
355 -                 Check(i)=1;
356 -                 CurHldA(i)=Cur;
357 -                 CurHldB(Cur)=i;
358 -                 CurHldC(Cur)=j;
359 -                 Cur=i;
360 -                 i=countBlk+3;
361 -                 break;
362 -             end
363
364 -         end
365 -     end
366
367 -     if i==countBlk+3
368
369 -         break;
370 -     end
371
372 -   end
373 -   if i==countBlk+2
374       %出力のない(子がない)NodeをSinkNodeにつなげる処理%
375 -     if (parCheck(Cur)==0 && task(Cur+1,2)==1) || (parCheck(Cur)==0 && LoopCheck(Cur)==1)
376 -         task(countBlk+3,2*task(countBlk+3,3)+4)=Cur;
377 -         task(countBlk+3,2*task(countBlk+3,3)+5)=0;
378 -         task(countBlk+3,3)=task(countBlk+3,3)+1;
379 -         parCheck(Cur)=parCheck(Cur)+1;
380 -         par(Cur,parCheck(Cur)+2)=countBlk+3;
381 -         par(Cur,2)=parCheck(Cur);
382 -     end
383
384 -     Check(Cur)=-1;   %訪問済み(それ以上進めない状態)
385 -     Cur=CurHldA(Cur);   %戻る
386 -   end
387
388 - end
389
390 - fid = fopen( 'result.txt', 'w+');
```

図 6.30: abstract.m（8）

```
391 -   fid2 = fopen('table.txt','w+');
392
393     %タスクグラフ作成
394 -   for i=1:countBlk+3
395 -       fprintf(fid,'% d', task(i,:) );
396 -       fprintf(fid,'¥r');
397 -   end
398     %対応表作成
399 -   fprintf(fid2,'%d:% s',1,'SrcNode'); %ソースノード書き込み
400 -   fprintf(fid2,'¥r¥n');
401 -   for i=1:countBlk
402 -        fprintf(fid2,'%d:% s',i+1,blkName{i});
403 -        fprintf(fid2,'¥r¥n');
404 -   end
405 -    fprintf(fid2,'%d:% s',countBlk+2,'SinkNode'); %シンクノード書き込み
406 -    fprintf(fid2,'¥r¥n');
407
408 -   fclose(fid);
409 -   fclose(fid2);
410
411     %disp('new');
412 -   disp(task);
413
414 -   disp('result.txtに保存しました。');
415 -   disp('table.txtに保存しました。');
416 -   clear;
417
```

図 6.31: abstract.m（9）

（２）次はブロック化法のプログラムを示す．本プログラムの実行ファイルはblock_construction.m という名前のスクリプト M-ファイルである．このプログラムは７つの関数が存在する．(i)「Chen」はタスクグラフの情報をファイルから読み込み，タスクグラフまた各ノードの構造体を構成する関数である．(ii)「List」は最遅開始時刻を用いたスケジューリング法に基づいてタスクグラフの優先リストを求める関数である．(iii)「Predecessor」は各ノードの先祖を求める関数である．(iv)「Successor」は各ノードの子孫を求める関数である．(v)「Block」はブロック化法中の基本ブロック化法を利用してタスクグラフ中のノードをブロックする関数である．(vi)「Newblock」はブロック化法中の追加ブロック化法を利用してタスクグラフ中のノードをブロックする関数である．本プログラムは基本ブロック化法は１回実行する．追加ブロック化法はブロックできる限り循環に実行することである．

```
1      function block_construction
2      %**************<<タスクグラフの各タスクの基本情報を構造体へ>>*****************
3      %(1)ファイルの読み込み（開始）
4  -   taskgraph_name=input('taskgraphの名前を入力してください：','s');
5  -   fd=fopen(taskgraph_name,'r');%ファイルを開く
6  -   line=fgets(fd);%第一行を読み込み（文字列）
7  -   array=str2num(line);%（文字列->数字）関数を使う
8  -   number_total=array(1);%配列の第一の変数を使う
9
10 -   datas=[];%空の配列を用意
11 -   for line_m=1:number_total+2;%ファイル中のすべてのデータを読み込み
12 -       line=fgets(fd);
13 -   temp=str2num(line);
14 -   datas=[datas;temp];%配列の追加
15 -   end
16     %disp(datas);%配列の表示***************************************************
17 -   fclose(fd);%ファイルの読み込み（終わり）
18 -   task_sample=[];
19 -   taskgraph=[];
20 -   groupoftaskgraph=[];
21 -   times=0;
22 -   exit_program=0;
23 -   graph=datas;
24 -   Chen
25 -   Predecessor
```

図 6.32: block_construction.m（１）

```
26 -   Successor
27 -   Block
28 -   while(1)
29 -   task_sample=[];
30 -   taskgraph=[];
31 -   Chen
32 -   List
33 -   Successor
34 -   Newblock
35     %if times==40;
36 -   if exit_program==1;%★★★★★★★チェック
37 -       disp('★★★★★このタスクグラフにブロック化法できない★★★');
38 -       disp('★★★実行できるブロックの数は"0"ので，プログラムそのまま終了★★★');
39 -       break;
40 -   end
41 -   times=times+1;
42 -   end
43 -   fprintf('追加ブロック化法の実行回数：( %d ) 回.¥n', times);
44     %check
45     %taskgraph
46 -   blocktable
47     %graph
48
49     %<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<入力>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
50     function Chen
51 -       taskgraph.total_task=number_total;%タスクの総数を求める（構造体に追加）
52         %number_total
53 -       array_size=zeros(1,2);%配列のサイズの配列を用意
54 -       array_size=size(graph);%サイズのデータの代入
55         %disp(array_size(1));disp(array_size(2));
56     %(2)タスクグラフの情報を表示（構造体）
57 -   for i=1:number_total+2;%配列中のデーターを一行一行で代入
58 -       line_x=1;%第2列（ノードの実行時間）
59 -       read_i=4;%第4列から（ノードの各親の番号と通信時間の情報）
60 -       ipread_i2=0;%ノードの各親の番号を求めるために
61 -       ipread_i3=0;%ノードの各通信時間を求めるために
62 -       for array_m=1:array_size(2);%データー配列の列の数
63 -           if line_x==1;
64 -               taskgraph.task(i).task_number=graph(i,array_m);
65 -               line_x=line_x+1;
66 -           elseif line_x==2;%ノードの実行時間を代入
67 -       taskgraph.task(i).exetime=graph(i,array_m);
68 -       line_x=line_x+1;
69 -           elseif line_x==3; %ノードの親の数の情報を代入
70 -       taskgraph.task(i).ip=graph(i,array_m);
71 -       line_x=line_x+1;
72 -           elseif line_x>3;
73 -               if read_i==line_x;%ノードの各親の番号情報を代入
74 -                   ipnumber_i=line_x-3;
75 -                   ipread_i=ipnumber_i-ipread_i2;
```

図 6.33: block_construction.m（2）

```
76              %taskgraph.task(i).ip_number(ipread_i)=datas(i,array_m);
77 -            task_sample(i).ip_number(ipread_i)=graph(i,array_m);
78 -            line_x=line_x+1;
79 -            ipread_i2=ipread_i2+1;
80 -        elseif read_i==line_x-1;%ノードの各通信時間の情報を代入
81 -            ipnumber_i=line_x-4;
82 -            ipread_i=ipnumber_i-ipread_i3;
83              %taskgraph.task(i).ip_read(ipread_i)=datas(i,array_m);
84 -            task_sample(i).ip_read(ipread_i)=graph(i,array_m);
85 -            line_x=line_x+1;
86 -            read_i=read_i+2;
87 -            ipread_i3=ipread_i3+1;
88 -        end
89 -      end
90 -    end
91 - end
92   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%<<<追加>>>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
93 -          for task_i=1:taskgraph.total_task+2;
94 -              for ip_i=1:taskgraph.task(task_i).ip;
95 -                  taskgraph.task(task_i).ip_number(ip_i)=task_sample(task_i).ip_number(ip_i);
96 -                  taskgraph.task(task_i).ip_read(ip_i)=task_sample(task_i).ip_read(ip_i);
97 -              end
98 -          end
99   %(3)タスクの子供の情報&&読み込み時間&&書き出す時間を求める（構造体情報の追加）
100 - for task_i=1:taskgraph.total_task+2;%各タスクの子と書き出し時間の初期化
101 -     taskgraph.task(task_i).is=0;
102 -     taskgraph.task(task_i).sum_write_time=0;
103 -     taskgraph.task(task_i).sum_read_time=0;
104 - end
105 - taskgraph.total_is=0;%タスクグラフの各タスクの親と子（初期化）
106 - taskgraph.total_ip=0;
107
108 - for task_i=2:taskgraph.total_task+2;%各タスクの子の情報を追加（構造体へ）
109 -     for ip_i=1:taskgraph.task(task_i).ip;
110 -         ipnumber_i=taskgraph.task(task_i).ip_number(ip_i);
111 -         is_i=taskgraph.task(ipnumber_i).is+1;
112 -         taskgraph.task(ipnumber_i).is_number(is_i)=task_i;
113 -         taskgraph.task(ipnumber_i).is=taskgraph.task(ipnumber_i).is+1;
114 -     end
115 - end
116 - for task_i=1:taskgraph.total_task+2;%タスクグラフの各タスクの親と子の総数
117 -     taskgraph.total_is=taskgraph.total_is+taskgraph.task(task_i).is;
118 -     taskgraph.total_ip=taskgraph.total_ip+taskgraph.task(task_i).ip;
119 - end
120 - for task_i=1:taskgraph.total_task+2;%各タスクの書き出し時間を求める（構造体へ）
121 -     for is_i=1:taskgraph.task(task_i).is;
122 -         is_number_temp=taskgraph.task(task_i).is_number(is_i);
123 -         for ip_i=1:taskgraph.task(is_number_temp).ip;
124 -             if taskgraph.task(is_number_temp).ip_number(ip_i)==task_i;
125 -                 taskgraph.task(task_i).sum_write_time=taskgraph.task(task_i).sum_write_time+taskgraph.task(is_number_temp).ip_read(ip_i);
```

図 6.34: block_construction.m（3）

```
126 -             break
127 -             end
128 -         end
129 -     end
130 - end
131 - for task_i=2:taskgraph.total_task+2;%各タスクの書き込み時間を求める（構造体へ）
132 -     for ip_i=1:taskgraph.task(task_i).ip;
133 -         taskgraph.task(task_i).sum_read_time=taskgraph.task(task_i).sum_read_time+taskgraph.task(task_i).ip_read(ip_i);
134 -     end
135 - end
136
137   %disp('各タスクの情報，総数：');%*******************************************
138   %disp(taskgraph.total_task+2);
139   %for i=1:taskgraph.total_task+2;
140   %disp(taskgraph.task(i));
141   %end
142   %disp(taskgraph);%**********************************************************
143 - end
144
145   %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
146
147
148   function List
149           %**********************<<優先リスト（デットライン法）>>*********************
150   %(4)各ノードの最長距離を求める
151 - task_dis=zeros(1,taskgraph.total_task+2);%total_task+2の長さの１元配列を用意する
152   %disp(taskgraph);
153   %disp(taskgraph.task(2));
154 - for task_i=1:taskgraph.total_task+2;%初期化（sノードからの距離&&各ノードのチェック）
155 -     task_dis(task_i)=0;
156 -     taskgraph.task(task_i).earliest=0;%★
157 -     check_task(task_i)=-1;
158 - end
159 - check_task(1)=0;%(sノードチェック完了)
160   %%%%%%%%%%%%%%%%%%%%(ip,is_flag前もって定義する,[改1])%%%%%%%%%%%%%%%%%%%%%%
161 - ip_flag=7;%        ||もしtaskの情報は何もない時に定義しなければ,エラーを出る||
162 - is_flag=7;%        ||「7」という数字は自分が好きだけだ                   ||
163   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
164
165 - while check_task(taskgraph.total_task+2)==-1;
166 -       for task_i=2:taskgraph.total_task+2;
167 -             if check_task(task_i)==-1;%探索中のノードがチェックしていない場合
168                 %task_i
169 -               for ip_i=1:taskgraph.task(task_i).ip;%すべての親によってチェックかどうかを確認する
170 -                   ip_flag=0;
171 -                   if check_task(taskgraph.task(task_i).ip_number(ip_i))==-1;%親がチェックしていない場合
172 -                       ip_flag=-1;
173 -                       break;
174 -                   end
175 -               end
```

図 6.35: block_construction.m（４）

```
176 -                       if ip_flag==0;%親がチェックした場合
177 -                           Maxdis=0;%最大距離の初期化
178 -                           for ip_i=1:taskgraph.task(task_i).ip;%すべての親によって最大距離を求める
179 -                               if  task_dis(taskgraph.task(task_i).ip_number(ip_i))>Maxdis;
180 -                                   Maxdis=task_dis(taskgraph.task(task_i).ip_number(ip_i));
181 -                               end
182 -                           end
183 -                           task_dis(task_i)=Maxdis+taskgraph.task(task_i).sum_read_time+taskgraph.task(task_i).exetime+
184 -                           taskgraph.task(task_i).sum_write_time;
185 -                           check_task(task_i)=0;
186 -                           taskgraph.task(task_i).earliest=Maxdis;
187 -                       end
188 -               end
189 -           end
190 -   end
191
192     %disp(task_dis);%各ノードの最長距離を確認************************************
193
194     %(5)各ノードの最遅開始時刻を求める(説明は(4)と同じ，省略)
195 -   time=zeros(1,taskgraph.total_task+2);%total_task+2の長さの１元配列を用意する
196 -   for task_i=1:taskgraph.total_task+2;%初期化 (各ノード最遅開始時刻&&各ノードのチェック)
197 -       time(task_i)=0;
198 -       check_task2(task_i)=-1;
199 -   end
200 -   check_task2(taskgraph.total_task+2)=0;%tノードチェック完了
201 -   time(taskgraph.total_task+2)=task_dis(taskgraph.total_task+2);%tノードの最遅開始時刻
202 -   while check_task2(1)==-1;
203 -   for task_i=1:taskgraph.total_task+1;
204 -       if check_task2(task_i)==-1;
205 -           for is_i=1:taskgraph.task(task_i).is;
206 -               is_flag=0;
207 -               if check_task2(taskgraph.task(task_i).is_number(is_i))==-1;
208 -                   is_flag=-1;
209 -                   break;
210 -               end
211 -           end
212 -           if is_flag==0;
213 -               Mintime=99999;
214 -               for is_i=1:taskgraph.task(task_i).is;
215 -                   if time(taskgraph.task(task_i).is_number(is_i))<Mintime;
216 -                       Mintime=time(taskgraph.task(task_i).is_number(is_i));
217 -                   end
218 -               end
219 -               time(task_i)=Mintime-taskgraph.task(task_i).sum_read_time-taskgraph.task(task_i).exetime-taskgraph.task(task_i).sum_write_time;
220 -               check_task2(task_i)=0;
221 -           end
222 -       end
223 -   end
224 -   end
225
```

図 6.36: block_construction.m（５）

```
226     %disp(time);%各ノードの最遅開始時刻を確認
227     %★★★★★★★(2013追加) (最遅開始時刻の情報が構造体に代入)
228 -   for task_i=1:taskgraph.total_task+2;
229 -       taskgraph.task(task_i).latest=time(task_i);
230 -       if taskgraph.task(task_i).latest==99999;%"Mintime"の数字と同じならば(17行前)
231 -           taskgraph.task(task_i).latest=0;
232 -       end
233 -   end
234     %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
235     %(6)優先リストを作成
236 -   task_list=zeros(1,taskgraph.total_task+2);%臨時リストを用意 (s, t を含む)
237     %time=sort(time)
238 -   for task_i=1:taskgraph.total_task+2;
239 -       task_list(task_i)=task_i;
240 -   end
241     %disp(task_list);
242 -   for task_i=2:taskgraph.total_task+1;%s, tノードを除く, ノードの昇順に並び
243 -       for task_i_temp=task_i+1:taskgraph.total_task+1;
244 -           if time(task_i)>time(task_i_temp);
245 -               Min=time(task_i);
246 -               time(task_i)=time(task_i_temp);
247 -               time(task_i_temp)=Min;
248 -               Min=task_list(task_i);
249 -               task_list(task_i)=task_list(task_i_temp);
250 -               task_list(task_i_temp)=Min;
251 -           end
252 -       end
253 -   end
254     %disp(task_list);
255 -   taskgraph.priority_list=zeros(1,taskgraph.total_task);%優先リストを用意
256 -   for task_i=1:taskgraph.total_task;
257 -       taskgraph.priority_list(task_i)=task_list(task_i+1);
258 -   end
259     %disp(taskgraph.priority_list);
260     %for task_i=1:taskgraph.total_task+2;
261     %    disp(taskgraph.task(task_i));
262     %end
263     %disp(taskgraph);
264 -   end
265
266     %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
267     function Block
268     %************************<<ブロック化法プログラム>>************************
269     %(10)ブロック化法中4つのパタンを確認する
270     %初期化(自分を含める先祖または子孫の配列を用意する)
271 -   for task_i=1:taskgraph.total_task+2;
272         %taskgraph.task(task_i).pre_number_temp=zeros(1:taskgraph.task(task_i).pre+1);%            自分を含める先祖配列用意(サイズ+1)
273 -       taskgraph.task(task_i).pre_number_temp=taskgraph.task(task_i).pre_number;%                   元先祖のデータを代入
274 -       taskgraph.task(task_i).pre_number_temp(taskgraph.task(task_i).pre+1)=task_i;%               配列中で最後の空白に自分の番号を入れる
275 -       taskgraph.task(task_i).pre_number_temp=sort(taskgraph.task(task_i).pre_number_temp);
```

図 6.37: block_construction.m（6）

```
276     %taskgraph.task(task_i).suc_number_temp=zeros(1:taskgraph.task(task_i).suc+1);%            自分を含める子孫配列用意(サイズ+1)
277 -   taskgraph.task(task_i).suc_number_temp=taskgraph.task(task_i).suc_number;%                 元子孫のデータを代入
278 -   taskgraph.task(task_i).suc_number_temp(taskgraph.task(task_i).suc+1)=task_i;%              配列中で最後の空白に自分の番号を入れる
279 -   taskgraph.task(task_i).suc_number_temp=sort(taskgraph.task(task_i).suc_number_temp);%       自分を含める子孫配列だけsortする
280 - end
281   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%改「2」%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
282   %||for関数中でそのようなzeros関数を使えない，上の5と8行のように（使ったらエラー)||
283   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
284 - total_group=0;     %groupの数を初期化
285 - group=[];          %groupという空の多次元配列を用意
286 - program_finish=1;  %★★★★★★★最初はプログラムは実行できると示す．
287   %disp(taskgraph.task(3));
288   %*****************************************************************************
289 - for task_i=2:taskgraph.total_task+1;     %ソースノードsが先祖がない，シングノードtが子孫がない．２つのノードを除く
290 -     for ip_i=1:taskgraph.task(task_i).ip;
291 -         ip_number_temp=taskgraph.task(task_i).ip_number(ip_i);
292 -         if ip_number_temp~=1;            %親がsノードじゃない場合，（sノードをブロックに含めないために)
293 -             flag_pre=1;                  %先祖確認の旗
294 -             flag_suc=1;                  %子孫確認の旗
295 -             for pre_i=1:taskgraph.task(task_i).pre;
296 -                 if taskgraph.task(task_i).pre~=taskgraph.task(ip_number_temp).pre+1;
297                      %このif分はmatlab限り，(matlab中で2つの要素比較する時にsize必ず同じ,c言語必要がない)
298 -                    flag_pre=0;
299 -                    break;
300 -                end
301 -                if taskgraph.task(task_i).pre_number(pre_i)~=taskgraph.task(ip_number_temp).pre_number_temp(pre_i);
302                      %親子関係の2つのノードが「先祖＋自分」＝「先祖」
303 -                   flag_pre=0;
304 -                   break;
305 -                end
306 -            end
307 -            for suc_i=1:taskgraph.task(task_i).suc;%先祖と同じ
308 -                if taskgraph.task(ip_number_temp).suc~=taskgraph.task(task_i).suc+1;%親子関係の2つのノードが配列size同じかどうか
309 -                    flag_suc=0;
310 -                    break;
311 -                end
312 -                if taskgraph.task(ip_number_temp).suc_number(suc_i)~=taskgraph.task(task_i).suc_number_temp(suc_i);
313                      %親子関係の2つのノードが「子孫」＝「子孫＋自分」
314 -                    flag_suc=0;
315 -                    break;
316 -                end
317 -            end
318 -            if flag_pre==1&&flag_suc==1;%2つの旗を確認したらgroupにいれる,(((group[groupの番号][1]:親,group[groupの番号][2]:自分)))
319 -                total_group=total_group+1;%group数を増えるように
320 -                temp=[ip_number_temp task_i];%各groupの仕組み
321 -                group=[group;temp];
322 -                break;
323 -            end
324 -        end
325 -    end
```

図 6.38: block_construction.m（７）

```
326 -   end
327     %disp(group);
328 -   out_group=group;
329 -   groupoftaskgraph=group;
330     %disp(group(1,2));
331     %disp(total_group);
332 -   if total_group==0;%★★★★★★★★groupの数のチェック
333 -       program_finish=0;
334 -       disp('★★★★★このタスクグラフにブロック化法できない★★★');
335 -       disp('★★★ブロックの数は"0"ので，プログラムそのまま終了★★★');
336 -   end
337     %%%%%%%%%%%%%%%%%%%%%%%%%各パターンの組み合わせ%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
338 -   if program_finish==1;%★★★★★★★★もし探したgroupの数は0にならば，プログラム終了(groupとは組み合わせしていないブロックです)
339 -   Complete=1;
340 -   InComplete=0;
341 -   OK=1;
342 -   NG=0;
343 -   total_block=0;%ブロックの数の初期化
344 -   for group_i=1:total_group;%[[[[[すべてgrouoの中身]]]]
345 -       block_temp(group_i).total=2;%各ブロックの中身の数の初期化
346 -       condition(group_i)=InComplete;%各パターンが実行したかどうか
347 -       search(group_i)=NG;%パターンがブロックに変更する条件
348 -   end
349 -   for task_i=1:taskgraph.total_task+2;%[[[[すべてタスクの中身]]]] (tableに含まれる各タスクの中身)
350 -       table_temp(task_i).total=1;
351 -   end
352
353 -   for group_i=1:total_group;
354 -           for group_i_temp=group_i+1:total_group;
355 -                   if group(group_i,1)==group(group_i_temp,1)&&group(group_i,2)==group(group_i_temp,2)
356 -                       condition(group_i_temp)=Complete;
357 -                   end
358 -           end
359 -   end
360
361 -   for group_i=1:total_group;
362 -        if condition(group_i)==InComplete;
363 -           while (1);
364 -                 finish(group_i)=block_temp(group_i).total;
365 -           for group_i_temp=group_i+1:total_group;
366 -               if condition(group_i_temp)==InComplete;
367                     %if group_i~=group_i_temp;
368 -                       for i=1:block_temp(group_i).total;
369 -                           if group(group_i,i)==group(group_i_temp,1);
370 -                            group(group_i,block_temp(group_i).total+1)=group(group_i_temp,2);%パターンに新たな内容を追加する
371 -                             block_temp(group_i).total=block_temp(group_i).total+1;
372 -                             condition(group_i_temp)=Complete;
373 -                           end
374 -                            if group(group_i,i)==group(group_i_temp,2);
375 -                                group(group_i,block_temp(group_i).total+1)=group(group_i_temp,1);%パターンに新たな内容を追加する
```

図 6.39: block_construction.m（８）

```
376 -                              block_temp(group_i).total=block_temp(group_i).total+1;
377 -                              condition(group_i_temp)=Complete;
378 -                          end
379 -                      end
380                       %end
381 -                 end
382 -             end
383 -             if finish(group_i)==block_temp(group_i).total;
384               %disp('確認');
385 -             break;
386 -             end
387 -         end
388 -         condition(group_i)=Complete;
389 -         search(group_i)=OK;
390 -     end
391 - end
392
393 - group1=[];
394 - for group_i=1:total_group;
395 -     if search(group_i)==OK;
396 -         group1(group_i,1)=group(group_i,1);
397 -             p=group(group_i,1);
398 -             j=0;
399 -                 for i=2:block_temp(group_i).total;
400 -                     if p==group(group_i,i);
401 -                         j=j+1;
402 -                     else
403 -                         group1(group_i,i-j)=group(group_i,i);
404 -                         p=group(group_i,i);
405 -                     end
406 -                 end
407 -                 block_temp(group_i).total=block_temp(group_i).total-j;
408 -     end
409 - end
410 - group=[];
411 - group=group1;
412
413   %group
414 - box=[];
415 - for group_i=1:total_group;
416 -     if search(group_i)==OK;
417 -         box_i=0;
418 -         for task_i=1:taskgraph.total_task+2;
419 -             check(task_i)=0;
420 -         end
421 -         for i=1:block_temp(group_i).total;
422 -             check(group(group_i,i))=1;
423 -         end
424 -         for i=1:block_temp(group_i).total;
425 -             for ip_i=1:taskgraph.task(group(group_i,i)).ip;
```

図 6.40: block_construction.m（９）

```
426 -                 check_flag=1;
427 -                 ipnumber_i=taskgraph.task(group(group_i,i)).ip_number(ip_i);
428 -                 if check(ipnumber_i)==1;
429 -                     check_flag=0;
430 -                     break;
431 -                 end
432 -             end
433
434 -             if check_flag==1;
435 -                 box_i=box_i+1;
436 -                 box(group_i,box_i)=group(group_i,i);
437 -                 break;
438 -             end
439 -         end
440
441 -         for i=1:block_temp(group_i).total;
442 -             if box(group_i,1)~=group(group_i,i);
443 -                 box_i=box_i+1;
444 -                 box(group_i,box_i)=group(group_i,i);
445 -             end
446 -         end
447
448 -     end
449 - end
450   %box
451
452 - group=box;
453
454 - for group_i=1:total_group;
455 -     if search(group_i)==OK;%条件を満たすパターンはブロックになる
456 -         total_block=total_block+1;
457 -             for block_i=1:block_temp(group_i).total;
458 -           block(total_block,block_i)=group(group_i,block_i);
459 -         end
460 -     end
461 - end
462 - fprintf('☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆¥n')
463 - fprintf('<<基本ブロック化法>>で得られたブロック：¥n')
464 - disp(block);
465 - fprintf('ブロックの総数：　(%d)　¥n',total_block)
466 - fprintf('☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆¥n')
467   %disp(total_block);
468   %(11)ブロックを新たなノードに変更%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
469   %初期化
470 - for task_i=1:taskgraph.total_task+2;
471 -     table(task_i,1)=task_i;
472       %table(task_i,2)=0;%★★★★★★★★%ブロック化法で実行したまま,更新できない場合,table配列の２番目は何も入っていないのでエラーでる.
473 - end
474 - for task_i=1:taskgraph.total_task+2;
475 -     flag(task_i)=NG;
```

図 6.41: block_construction.m（１０）

```
476 -    end
477 -    for group_i=1:total_group;
478 -        if search(group_i)==OK;
479 -            for block_i=2:block_temp(group_i).total;
480 -                flag(group(group_i,block_i))=OK;
481 -            end
482 -        end
483 -    end
484      %disp(flag(group(3,3)));
485      %%%%%%%%%%%%%%%%%%%%%%%%%%%%tableを作る（準備作業）<1>%%%%%%%%%%%%%%%%%%%%%%%%%
486 -    for task_i=1:taskgraph.total_task+2;
487 -        task_change(task_i,1)=task_i;
488 -        task_change(task_i,2)=task_i;
489 -    end
490 -    task_temp=taskgraph.total_task+2;
491 -    for group_i=1:total_group;
492 -        if search(group_i)==OK;
493 -            task_temp=task_temp+1;
494 -            for block_i=1:block_temp(group_i).total;
495 -                task_change(group(group_i,block_i),2)=task_temp;
496 -            end
497 -        end
498 -    end
499      %disp(task_change);
500 -    for task_i=1:taskgraph.total_task+2;
501 -        if task_change(task_i,1)~=task_change(task_i,2)&&flag(task_change(task_i,1))==OK;
502 -            table(task_i,1)=-1;
503 -        end
504 -    end
505 -    for task_i=1:taskgraph.total_task+2;
506 -        for group_i=1:total_group;
507 -            if search(group_i)==OK;
508 -                if table(task_i,1)==group(group_i,1);
509 -                    for block_i=1:block_temp(group_i).total;
510 -                        table(task_i,block_i+1)=group(group_i,block_i);
511 -                    end
512 -                    table_temp(task_i).total=block_i+1;
513 -                end
514 -            end
515 -        end
516 -    end
517      %disp(table);
518      %for task_i=1:taskgraph.total_task+2;
519      %disp(table_temp(task_i).total);
520      %end
521      %%%%%%%%%%%%%%%ブロックが新たなノードに変換し，その親の情報<2>%%%%%%%%%%%%%%%%%%%
522 -    for task_i=1:taskgraph.total_task+2;
523 -        condition(task_i)=NG;
524 -    end
525 -    for task_i=1:taskgraph.total_task+2;
```

図 6.42: block_construction.m（１１）

```
526 -     if table(task_i,2)~=0;
527 -         condition(table(task_i,2))=OK;
528 -         for ip_i=1:taskgraph.task(table(task_i,2)).ip;
529 -             condition(taskgraph.task(table(task_i,2)).ip_number(ip_i))=OK;
530 -         end
531 -         for i=3:table_temp(task_i).total;
532 -             condition(table(task_i,i))=OK;
533 -         end
534 -         for i=3:table_temp(task_i).total;
535 -             taskgraph.task(table(task_i,2)).exetime=taskgraph.task(table(task_i,2)).exetime+taskgraph.task(table(task_i,i)).exetime;
536 -             for ip_i_temp=1:taskgraph.task(table(task_i,i)).ip;
537 -                 if condition(taskgraph.task(table(task_i,i)).ip_number(ip_i_temp))~=OK;
538 -                     ip_i=ip_i+1;
539 -                     taskgraph.task(table(task_i,2)).ip_number(ip_i)=taskgraph.task(table(task_i,i)).ip_number(ip_i_temp);
540 -                     taskgraph.task(table(task_i,2)).ip_read(ip_i)=taskgraph.task(table(task_i,i)).ip_read(ip_i_temp);
541                       %taskgraph.task(table(task_i,2)).sum_read_time=taskgraph.task(table(task_i,2)).sum_read_time+
542 -                     taskgraph.task(table(task_i,2)).ip_read(ip_i);
543 -                     taskgraph.task(table(task_i,2)).ip=taskgraph.task(table(task_i,2)).ip+1;
544 -                     condition(taskgraph.task(table(task_i,i)).ip_number(ip_i_temp))=OK;
545 -                 end
546 -             end
547 -         end
548 -         for i=2:table_temp(task_i).total;
549 -             condition(table(task_i,i))=NG;
550 -         end
551 -         for ip_i=1:taskgraph.task(task_i).ip;
552 -             condition(taskgraph.task(task_i).ip_number(ip_i))=NG;
553 -         end
554 -     end
555 - end
556
557   %for task_i=1:taskgraph.total_task+2;
558   %    disp(taskgraph.task(task_i).exetime);
559   %end
560   %           disp('確認');
561   %           disp(5);
562   %           disp(taskgraph.task(table(5,2)).exetime);
563   %           disp(taskgraph.task(table(5,3)).exetime);
564   %%%%%%%%%%%%%%%ブロックが新たなノードに変換し，その子の情報<3>%%%%%%%%%%%%%%%%%%%[******注意******]このプログラムなくでもいい(188~218)
565 - for task_i=1:taskgraph.total_task+2;
566 -     if table(task_i,2)~=0;
567 -         condition(table(task_i,table_temp(task_i).total))=OK;
568 -         i=table_temp(task_i).total-1;
569 -         for is_i=1:taskgraph.task(table(task_i,table_temp(task_i).total)).is;
570 -             condition(taskgraph.task(table(task_i,table_temp(task_i).total)).is_number(is_i))=OK;
571 -         end
572 -         while i>1;
573 -             condition(table(task_i,i))=OK;
574 -             for is_i_temp=1:taskgraph.task(table(task_i,i)).is;
575 -                 if condition(taskgraph.task(table(task_i,i)).is_number(is_i_temp))~=OK;
```

図 6.43: block_construction.m（１２）

```
576 -                      is_i=is_i+1;
577 -                      taskgraph.task(table(task_i,table_temp(task_i).total)).is_number(is_i)=
578 -                      taskgraph.task(table(task_i,i)).is_number(is_i_temp);
579 -                      taskgraph.task(table(task_i,table_temp(task_i).total)).is=
580 -                      taskgraph.task(table(task_i,table_temp(task_i).total)).is+1;
581 -                      condition(taskgraph.task(table(task_i,i)).is_number(is_i_temp))=OK;
582 -                  end
583 -              end
584 -              i=i-1;
585 -          end
586 -          for i=1:table_temp(task_i).total;
587 -              condition(table(task_i,i))=NG;
588 -          end
589 -          for is_i=1:taskgraph.task(task_i).is;
590 -              condition(taskgraph.task(task_i).is_number(is_i))=NG;
591 -          end
592 -      end
593 -  end
594    %for task_i=1:taskgraph.total_task+2;
595    %    disp(taskgraph.task(task_i));
596    %end
597    %%%%%%%%%%%%%%%%%%%%%%%%%%%%子の最後のデータを最初に移動<4>%%%%%%%%%%%%%%%%%%%%%%%%%%[******注意******]このプログラムなくでもいい(219~230)
598 -  for task_i=1:taskgraph.total_task+2;
599 -      if table(task_i,2)~=0;
600 -          taskgraph.task(table(task_i,2)).is=taskgraph.task(table(task_i,table_temp(task_i).total)).is;
601 -          for is_i=1:taskgraph.task(table(task_i,table_temp(task_i).total)).is;
602 -              taskgraph.task(table(task_i,2)).is_number(is_i)=taskgraph.task(table(task_i,table_temp(task_i).total)).is_number(is_i);
603 -          end
604 -      end
605 -  end
606    %for task_i=1:taskgraph.total_task+2;
607    %    disp(taskgraph.task(task_i));
608    %end
609    %%%%%%%%%%%%%%%%%%%%%%%%%%%%すべてのノードの親の情報の更新<5>%%%%%%%%%%%%%%%%%%%%%%%%
610 -  for task_i=1:taskgraph.total_task+2;
611 -      for ip_i=1:taskgraph.task(task_i).ip;
612 -          for task_i_temp=1:taskgraph.total_task+2;
613 -              if table(task_i_temp,2)~=0;
614 -                  for i=3:table_temp(task_i_temp).total;
615 -                      if taskgraph.task(task_i).ip_number(ip_i)==table(task_i_temp,i);
616 -                          taskgraph.task(task_i).ip_number(ip_i)=table(task_i_temp,2);
617 -                      end
618 -                  end
619 -              end
620 -          end
621 -      end
622 -  end
623    %for task_i=1:taskgraph.total_task+2;
624    %    disp(taskgraph.task(task_i));
625    %end
```

図 6.44: block_construction.m（１３）

```
626      %%%%%%%%%%%%%%%%%%%%%%%%%%%すべてのノードの子の情報の更新<6>%%%%%%%%%%%%%%%%%%%%%%%%[******注意******]このプログラムなくてもいい(248~264)
627 -    for task_i=1:taskgraph.total_task+2;
628 -        for is_i=1:taskgraph.task(task_i).is;
629 -            for task_i_temp=1:taskgraph.total_task+2;
630 -                if table(task_i_temp,2)~=0;
631 -                    for i=3:table_temp(task_i_temp).total;
632 -                        if taskgraph.task(task_i).is_number(is_i)==table(task_i_temp,i);
633 -                            taskgraph.task(task_i).is_number(is_i)=table(task_i_temp,2);
634 -                        end
635 -                    end
636 -                end
637 -            end
638 -        end
639 -    end
640      %for task_i=1:taskgraph.total_task+2;
641      %    disp(taskgraph.task(task_i));
642      %end
643      %%%%%%%%%%%%%%%%%%%%%%%%%%%要らないノードの情報を捨てる<7>%%%%%%%%%%%%%%%%%%%%%%%%
644 -    for task_i=1:taskgraph.total_task+2;
645 -        if table(task_i,1)==-1;
646              %taskgraph.task(task_i).task_number=0;
647 -            taskgraph.task(task_i).exetime=0;
648 -            taskgraph.task(task_i).ip=0;
649 -            taskgraph.task(task_i).ip_number=0;
650 -            taskgraph.task(task_i).ip_read=0;
651 -            taskgraph.task(task_i).is=0;
652 -            taskgraph.task(task_i).sum_write_time=0;
653 -            taskgraph.task(task_i).sum_read_time=0;
654 -            taskgraph.task(task_i).is_number=0;
655 -            taskgraph.task(task_i).pre=0;
656 -            taskgraph.task(task_i).pre_number=0;
657 -            taskgraph.task(task_i).suc=0;
658 -            taskgraph.task(task_i).suc_number=0;
659 -            taskgraph.task(task_i).pre_number_temp=0;
660 -            taskgraph.task(task_i).suc_number_temp=0;
661 -        end
662 -    end
663      %for task_i=1:taskgraph.total_task+2;
664      %    disp(taskgraph.task(task_i));
665      %end
666      %%%%%%%%%%%%%%%%%%%%%%%%%%%親と子のデータの並び（昇）<8>%%%%%%%%%%%%%%%%%%%%%%%%
667 -    for task_i=1:taskgraph.total_task+2;
668 -        for ip_i=1:taskgraph.task(task_i).ip;
669 -            min_no=ip_i;
670 -            min_number=taskgraph.task(task_i).ip_number(ip_i);
671 -            min_read=taskgraph.task(task_i).ip_read(ip_i);
672 -            for ip_i_temp=ip_i+1:taskgraph.task(task_i).ip;
673 -                if taskgraph.task(task_i).ip_number(ip_i_temp)<min_number;
674 -                    min_no=ip_i_temp;
675 -                    min_number=taskgraph.task(task_i).ip_number(ip_i_temp);
```

図 6.45: block_construction.m（１４）

```
676 -                 min_read=taskgraph.task(task_i).ip_read(ip_i_temp);
677 -             end
678 -         end
679 -         taskgraph.task(task_i).ip_number(min_no)=taskgraph.task(task_i).ip_number(ip_i);
680 -         taskgraph.task(task_i).ip_read(min_no)=taskgraph.task(task_i).ip_read(ip_i);
681 -         taskgraph.task(task_i).ip_number(ip_i)=min_number;
682 -         taskgraph.task(task_i).ip_read(ip_i)=min_read;
683 -     end
684 - end
685   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
686 - for task_i=1:taskgraph.total_task+2;
687 -     taskgraph.task(task_i).is_number=sort(taskgraph.task(task_i).is_number);
688 - end
689   %%%%%%%%%%%%%%%%%%%%%%%%%%%同じ結果を削除(子)<9>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%[******注意******]このプログラムなくでもいい(294~311)
690 - for task_i=1:taskgraph.total_task+1;
691 -     if taskgraph.task(task_i).is~=0;%★★★★★★★★★★★★★★★★★★★★★情報なしのノード情報が実行しない★★★★★★★★★★★★★★★★★★
692 -     p=taskgraph.task(task_i).is_number(1);
693 -     task_sample(task_i).is_number_temp(1)=taskgraph.task(task_i).is_number(1);
694 -     j=0;%削除した数
695 -     for i=2:taskgraph.task(task_i).is;
696 -         if p==taskgraph.task(task_i).is_number(i);
697 -             j=j+1;
698 -         else
699 -              task_sample(task_i).is_number_temp(i-j)=taskgraph.task(task_i).is_number(i);
700 -              p=taskgraph.task(task_i).is_number(i);
701 -         end
702 -     end
703 -     taskgraph.task(task_i).is=taskgraph.task(task_i).is-j;
704 -     end
705 - end
706   %task_sample
707 - for task_i=1:taskgraph.total_task+1;
708 - taskgraph.task(task_i).is_number=[];
709 - end
710 - for task_i=1:taskgraph.total_task+1;
711 -     taskgraph.task(task_i).is_number=task_sample(task_i).is_number_temp;
712 - end
713
714   %%%%%%%%%%%%%%%%%%%%%%%%%%%同じ結果を削除(親)<10>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
715 - for task_i=2:taskgraph.total_task+2;
716 -     if taskgraph.task(task_i).ip~=0;%★★★★★★★★★★★★★★★★★★★★★情報なしのノード情報が実行しない★★★★★★★★★★★★★★★★★★
717 -     p=taskgraph.task(task_i).ip_number(1);
718 -     task_sample(task_i).ip_number_temp(1)=taskgraph.task(task_i).ip_number(1);
719 -     task_sample(task_i).ip_read_temp(1)=taskgraph.task(task_i).ip_read(1);
720 -     j=0;%削除した数
721 -     for i=2:taskgraph.task(task_i).ip;
722 -         if p==taskgraph.task(task_i).ip_number(i);
723 -             j=j+1;
724 -         else
725 -             task_sample(task_i).ip_number_temp(i-j)=taskgraph.task(task_i).ip_number(i);
```

図 6.46: block_construction.m（１５）

```
726 -                 task_sample(task_i).ip_read_temp(i-j)=taskgraph.task(task_i).ip_read(i);
727 -                 p=taskgraph.task(task_i).ip_number(i);
728 -             end
729 -         end
730 -         taskgraph.task(task_i).ip=taskgraph.task(task_i).ip-j;
731 -         end%%%%%%%%%%%%%%%%%%%%%%%%%%%%
732 -     end
733 -     for task_i=2:taskgraph.total_task+2;
734 -         taskgraph.task(task_i).ip_number=[];
735 -         taskgraph.task(task_i).ip_read=[];
736 -     end
737 -     for task_i=2:taskgraph.total_task+2;
738 -         taskgraph.task(task_i).ip_number=task_sample(task_i).ip_number_temp;
739 -         taskgraph.task(task_i).ip_read=task_sample(task_i).ip_read_temp;
740 -     end
741
742       %disp(taskgraph.task(2));
743       %for task_i=1:taskgraph.total_task+2;
744       %    disp(taskgraph.task(task_i));
745       %end
746       %%%%%%%%%%%%%%%%%%%%%%ブロック後のtaskgraphデータを整理する<11>%%%%%%%%%%%%%%%%%
747 -     graph=[];
748       %graph(1,1)=taskgraph.total_task;
749 -     for graph_i=1:taskgraph.total_task+2;%sとtとgraph(1,1)
750 -         graph(graph_i,1)=taskgraph.task(graph_i).task_number;
751 -         graph(graph_i,2)=taskgraph.task(graph_i).exetime;
752 -         graph(graph_i,3)=taskgraph.task(graph_i).ip;
753 -         graph_j=4;
754 -         for ip_i=1:taskgraph.task(graph_i).ip;
755 -             graph(graph_i,graph_j)=taskgraph.task(graph_i).ip_number(ip_i);
756 -             graph_j=graph_j+1;
757 -             graph(graph_i,graph_j)=taskgraph.task(graph_i).ip_read(ip_i);
758 -             graph_j=graph_j+1;
759 -         end
760 -     end
761       %graph
762 -     end%★★★★★★★program_finishのend
763 -     end
764
765
766       %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
767       function Newblock
768       %************************<<新ブロック化法プログラム>>**********************
769       %(12)新ブロック化法%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
770 -     group=[];
771 -     total_group=0;
772 -     program_finish=1;  %★★★★★★★★最初はプログラムは実行できると示す.
773       %          ☆開始
774 -     for task_i=2:taskgraph.total_task+1;     %ソースノードsが親がない, シングノードtが子がない. ２つのノードを除く
775       %★★★★★★★★★★★★★★★★<<<パターン1,3開始>>>★★★★★★★★★★★★★★★★
```

図 6.47: block_construction.m（１６）

```
776 -     if taskgraph.task(task_i).ip>1;      %親が複数の場合(パターン１)(パターン３)
777 -       Maxearliest=-1;                    %選ばれたノードの最大の最早開始時刻の初期化
778   %       ☆親が複数持ってるノードの親の最大の最早開始時刻を求め
779 -       for ip_i=1:taskgraph.task(task_i).ip;
780 -           ip_number_temp=taskgraph.task(task_i).ip_number(ip_i);
781 -           if taskgraph.task(ip_number_temp).earliest>Maxearliest;
782 -               Maxearliest=taskgraph.task(ip_number_temp).earliest;
783 -           end
784 -       end
785         %disp(Maxearliest);
786   %       ☆最大の最早開始時刻を持っている親がどうの親のことを確認
787 -       for ip_i=1:taskgraph.task(task_i).ip;
788 -           flag1=0;%しるしの初期化
789 -           flag3=0;
790 -           flag3_temp=0;
791 -           ip_number_temp=taskgraph.task(task_i).ip_number(ip_i);
792 -           if ip_number_temp~=1;%親ノードがソースノードじゃない場合
793 -               if Maxearliest==taskgraph.task(ip_number_temp).earliest;
794 -                   number=ip_i;%最大の最早開始時刻を持っている親の番号--->number
795                     %task_i                   %(選ばれるノード番号)
796                     %disp(ip_number_temp);    %(最大の最早開始時刻を持っている親番号)
797   %---------------->パターン１開始
798 -                   if taskgraph.task(ip_number_temp).is==1;%★★★★★★★(パターン1の場合)★★★★★★★
799 -                       flag1=1;                   %パターン１の実行できるかどうかの確認印，まずブロックできると示す
800   %                   ☆親の中で選ばれた親ノードと別の親を調べてその条件が満たすかどうか
801 -                   for ip_j=1:taskgraph.task(task_i).ip;%ノードの親の中で
802 -                       ip_number_j=taskgraph.task(task_i).ip_number(ip_j);
803 -                       if ip_number_j~=ip_number_temp;%自分じゃない場合
804 -                       if taskgraph.task(ip_number_temp).latest<taskgraph.task(ip_number_j).earliest+
805 -                           taskgraph.task(ip_number_j).sum_read_time+taskgraph.task(ip_number_j).exetime+
806 -                           taskgraph.task(ip_number_j).sum_write_time;
807 -                           flag1=0;      %以上の式満たしたらパターン１がブロックできない
808 -                           break;        %もし任意の１つの親の条件が満たさなければ，次の親を調べる必要がない(30行のfor文を出す)
809 -                       end
810 -                       end
811 -                   end
812 -                   end%pattern1終止
813   %---------------->パターン１終わり
814   %---------------->パターン３開始
815 -                   if taskgraph.task(ip_number_temp).is>1;%★★★★★★★(パターン3の場合)★★★★★★★
816 -                       flag3=1;
817 -                       flag3_temp=1;
818   %                       ☆親と別の親の調べ
819 -                       for ip_j=1:taskgraph.task(task_i).ip;
820 -                           ip_number_j=taskgraph.task(task_i).ip_number(ip_j);
821 -                           if ip_number_j~=ip_number_temp;%自分じゃない場合
822                                 %task_i
823                                 %ip_number_temp
824 -                               if taskgraph.task(ip_number_temp).latest<taskgraph.task(ip_number_j).earliest+
825 -                                   taskgraph.task(ip_number_j).sum_read_time+taskgraph.task(ip_number_j).exetime+
```

図 6.48: block_construction.m（１７）

```
826 -                      taskgraph.task(ip_number_j).sum_write_time;
827 -                     flag3_temp=0;     %以上の式満たしたらパターン3がブロックできない
828 -                     break;       %任意の親が条件を満たさなければ，for文を出す
829 -                  end
830 -               end
831 -            end
832 -            if flag3_temp~=1%もし以上の条件を満たさなければ，pattern3が実行できない
833 -               flag3=0;
834 -            end
835   %          ☆もし条件を満たしたら，次ノードと別の子と調べ
836 -            if flag3_temp==1;
837 -               for is_j=1:taskgraph.task(ip_number_temp).is;
838 -                   is_number_j=taskgraph.task(ip_number_temp).is_number(is_j);
839 -                   if is_number_j~=task_i;%自分じゃない場合
840 -                      if taskgraph.task(is_number_j).latest<taskgraph.task(ip_number_temp).earliest+
841 -                          taskgraph.task(ip_number_temp).sum_read_time+taskgraph.task(ip_number_temp).exetime+
842 -                          taskgraph.task(ip_number_temp).sum_write_time+taskgraph.task(task_i).sum_read_time+
843 -                          taskgraph.task(task_i).exetime+taskgraph.task(task_i).sum_write_time-
844 -                          2*taskgraph.task(task_i).ip_read(number);
845 -                          flag3=0;     %以上の式満たしたらパターン2がブロックできない
846 -                         break;
847 -                      end
848 -                   end
849 -               end
850 -            end
851 -         end%pattern3終止
852   %---------------->パターン３終わり
853 -            break;
854             %最大の最早開始時刻を持っている親の番号が確認終わったら，次の調べるのことが必要がない，
855             |(特別：親の最大が同じの場合)
856 -         end
857 -      end
858 -   end
859     %task_i
860      %ip_number_temp
861      %flag3
862
863   %<<<パターン1>>>のデータのまとめ
864 -      if flag1==1;               %もし以上の式を満たさなければ，その2つのノードがブロックできる
865 -         total_group=total_group+1;
866 -         temp1=[ip_number_temp task_i];
867 -         group=[group;temp1];
868 -      end
869
870   %<<<パターン3>>>のデータのまとめ
871 -      if flag3==1;               %もし以上の式を満たさなければ，その2つのノードがブロックできる
872           %task_i
873           %disp('確認3');
874 -         total_group=total_group+1;
875 -         temp3=[ip_number_temp task_i];
```

図 6.49: block_construction.m（１８）

```
876 -            group=[group;temp3];
877 -         end
878 -     end%pattern(1,3)の終止
879    %★★★★★★★★★★★★★★★★<<<パターン2,4開始>>>★★★★★★★★★★★★★★★★
880 -     if taskgraph.task(task_i).is>1;      %子が複数の場合(パターン2)(パターン4)
881 -        Minlatest=99999;                   %選ばれたノードの最小の最遅開始時刻の初期化
882    %       ☆子が複数持ってるノードの子の最小の最遅開始時刻を求め
883 -        for is_i=1:taskgraph.task(task_i).is;
884 -            is_number_temp=taskgraph.task(task_i).is_number(is_i);
885 -            if taskgraph.task(is_number_temp).latest<Minlatest;
886 -                Minlatest=taskgraph.task(is_number_temp).latest;
887 -            end
888 -        end
889         %disp(Minlatest);
890     %      ☆最小の最遅開始時刻を持っている子がどうの子のことを確認
891 -        for is_i=1:taskgraph.task(task_i).is;
892 -            flag2=0;%しるしの初期化
893 -            flag4=0;
894 -            flag4_temp=0;
895 -            is_number_temp=taskgraph.task(task_i).is_number(is_i);
896 -            if is_number_temp~=taskgraph.total_task+2;%子ノードがシングノードじゃない場合
897 -               if Minlatest==taskgraph.task(is_number_temp).latest;
898                   %number=is_i;%最小の最遅開始時刻を持っている子の番号--->number
899                   %task_i                      %(選ばれるノード番号)
900                   %disp(is_number_temp);     %(最小の最遅開始時刻を持っている子番号)
901    %----------------->パターン２開始
902 -                 if taskgraph.task(is_number_temp).ip==1;%★★★★★★★(パターン2の場合)★★★★★★★
903 -                    flag2=1;                 %パターン2の実行できる確認印
904    %                 ☆子の中で選ばれた子ノードと別の子を調べてその条件が満たすかどうか
905 -                 for is_j=1:taskgraph.task(task_i).is;%ノードの子の中で
906 -                     is_number_j=taskgraph.task(task_i).is_number(is_j);
907 -                     if is_number_j~=is_number_temp;%自分じゃない場合
908 -                     if taskgraph.task(is_number_j).latest<taskgraph.task(task_i).earliest+
909 -                         taskgraph.task(task_i).sum_read_time+taskgraph.task(task_i).exetime+
910 -                         taskgraph.task(task_i).sum_write_time+taskgraph.task(is_number_temp).exetime+
911 -                         taskgraph.task(is_number_temp).sum_write_time-taskgraph.task(is_number_temp).sum_read_time;
912 -                         flag2=0;       %以上の式満たしたらパターン2がブロックできない
913 -                         break;         %もし任意の１つの親の条件が満たさなければ，次の子を調べる必要がない(for文を出す)
914 -                     end
915 -                     end
916 -                 end
917 -                   end%pattern2終止
918    %------------------>パターン２終わり
919    %------------------>パターン４開始
920 -                 if taskgraph.task(is_number_temp).ip>1;%★★★★★★★(パターン4の場合)★★★★★★★
921 -                     for ip_i=1:taskgraph.task(is_number_temp).ip;
922 -                        if task_i==taskgraph.task(is_number_temp).ip_number(ip_i);
923 -                            number=ip_i;
924 -                            break;
925 -                        end
```

図 6.50: block_construction.m（１９）

```
926 -              end
927 -              flag4=1;
928 -              flag4_temp=1;
929    %           ☆子と別の子の調べ
930 -              for is_j=1:taskgraph.task(task_i).is;
931 -                  is_number_j=taskgraph.task(task_i).is_number(is_j);
932 -                  if is_number_j~=is_number_temp;%自分じゃない場合
933 -                      if taskgraph.task(is_number_j).latest<taskgraph.task(task_i).earliest+
934 -                          taskgraph.task(task_i).sum_read_time+taskgraph.task(task_i).exetime+
935 -                          taskgraph.task(task_i).sum_write_time+taskgraph.task(is_number_temp).sum_read_time+
936 -                          taskgraph.task(is_number_temp).exetime+taskgraph.task(is_number_temp).sum_write_time-
937 -                          2*taskgraph.task(is_number_temp).ip_read(number);
938 -                          flag4_temp=0;      %以上の式満たしたらパターン4がブロックできない
939 -                          break;       %任意の子が条件を満たさなければ, for文を出す
940 -                      end
941 -                  end
942 -              end
943 -              if flag4_temp~=1%もし以上の条件を満たさなければ, pattern4が実行続けできない
944 -                  flag4=0;
945 -              end
946    %           ☆もし条件を満たしたら, 次ノードと別の親と調べ
947 -              if flag4_temp==1;
948 -                  for ip_j=1:taskgraph.task(is_number_temp).ip;
949 -                      ip_number_j=taskgraph.task(is_number_temp).ip_number(ip_j);
950 -                      if ip_number_j~=task_i;%自分じゃない場合
951 -                          if  taskgraph.task(task_i).latest<taskgraph.task(ip_number_j).earliest+
952 -                              taskgraph.task(ip_number_j).sum_read_time+taskgraph.task(ip_number_j).exetime+
953
954 -                              taskgraph.task(ip_number_j).sum_write_time;
955 -                              flag4=0;      %以上の式満たしたらパターン2がブロックできない
956 -                             break;
957 -                          end
958 -                      end
959 -                  end
960 -              end
961 -          end%pattern3終止
962    %----------------->パターン４終わり
963 -                break;
964                 %最小の最遅開始時刻を持っている子の番号が確認終わったら, 次の調べるのことが必要がない,
965                 |(特別：子の最小が同じの場合)
966 -         end
967 -     end
968 -   end
969    %<<<パターン2>>>のデータのまとめ
970 -   if  flag2==1; %もし以上の式を満たさなければ, その2つのノードがブロックできる
971         %disp('確認');
972 -       total_group=total_group+1;
973 -       temp2=[task_i is_number_temp];
974 -       group=[group;temp2];
975 -   end
```

図 6.51: block_construction.m（２０）

```
976     %<<<パターン4>>>のデータのまとめ
977 -         if flag4==1;                    %もし以上の式を満たさなければ，その2つのノードがブロックできる
978              %task_i
979              %disp('確認4');
980 -            total_group=total_group+1;
981 -            temp4=[task_i is_number_temp];
982 -            group=[group;temp4];
983 -         end
984 -   end%pattern(2,4)終止
985 -   end%taskgraphの終止
986     %disp(group);
987 -   out_group=group;
988     %total_group
989     %blockを作る
990 -   if total_group==0;%★★★★★★★groupの数のチェック
991 -       program_finish=0;
992 -       disp('★★★★★このタスクグラフにブロック化法できない★★★');
993 -       disp('★★★ブロックの数は"0"ので，プログラムそのまま終了★★★');
994 -   end
995     %%%%%%%%%%%%%%%%%%%%%%%%%各パターンの組み合わせ%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
996 -   if program_finish==1;%★★★★★★★もし探したgroupの数は0にならば，プログラム終了(groupとは組み合わせしていないブロックです)
997 -   Complete=1;
998 -   InComplete=0;
999 -   OK=1;
1000 -  NG=0;
1001 -  total_block=0;%ブロックの数の初期化
1002 -  for group_i=1:total_group;%[[[[[すべてgrouoの中身]]]]
1003 -      block_temp(group_i).total=2;%各ブロックの中身の数の初期化
1004 -      condition(group_i)=InComplete;%各パターンが実行したかどうか
1005 -      search(group_i)=NG;%パターンがブロックに変更する条件
1006 -  end
1007 -  for task_i=1:taskgraph.total_task+2;%[[[[すべてタスクの中身]]]] (tableに含まれる各タスクの中身
1008 -      table_temp(task_i).total=1;
1009 -  end
1010
1011    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
1012 -  for group_i=1:total_group;
1013 -          for group_i_temp=group_i+1:total_group;
1014 -                  if group(group_i,1)==group(group_i_temp,1)&&group(group_i,2)==group(group_i_temp,2)
1015 -                      condition(group_i_temp)=Complete;
1016 -                  end
1017 -          end
1018 -  end
1019
1020 -  for group_i=1:total_group;
1021 -      if condition(group_i)==InComplete;
1022 -          while (1);
1023 -               finish(group_i)=block_temp(group_i).total;
1024 -          for group_i_temp=group_i+1:total_group;
1025 -              if condition(group_i_temp)==InComplete;
```

図 6.52: block_construction.m（２１）

```
1026 -                  for i=1:block_temp(group_i).total;
1027 -                      if group(group_i,i)==group(group_i_temp,1);
1028                            %for i1=1:block_temp(group_i).total;
1029                             %   if group(group_i,i1)~=group(group_i_temp,2);
1030 -                           group(group_i,block_temp(group_i).total+1)=group(group_i_temp,2);%パターンに新たな内容を追加する
1031 -                           block_temp(group_i).total=block_temp(group_i).total+1;
1032 -                           condition(group_i_temp)=Complete;
1033                              %  end
1034                            %end
1035 -                      end
1036 -                       if group(group_i,i)==group(group_i_temp,2);
1037 -                           group(group_i,block_temp(group_i).total+1)=group(group_i_temp,1);%パターンに新たな内容を追加する
1038 -                           block_temp(group_i).total=block_temp(group_i).total+1;
1039 -                           condition(group_i_temp)=Complete;
1040 -                       end
1041 -                  end
1042                  %end
1043 -              end
1044 -          end
1045 -          if finish(group_i)==block_temp(group_i).total;
1046 -          break;
1047 -          end
1048 -          end
1049 -          condition(group_i)=Complete;
1050 -          search(group_i)=OK;
1051 -      end
1052 -  end
1053
1054
1055 -  group1=[];%ブロックに同じ数字があったら削除する例えば (1,2,3,4,2,2)
1056 -  for group_i=1:total_group;
1057 -      if search(group_i)==OK;
1058 -          group1(group_i,1)=group(group_i,1);
1059 -              p=group(group_i,1);
1060 -              j=0;
1061 -                  for i=2:block_temp(group_i).total;
1062 -                      if p==group(group_i,i);
1063 -                          j=j+1;
1064 -                      else
1065 -                          group1(group_i,i-j)=group(group_i,i);
1066 -                          p=group(group_i,i);
1067 -                      end
1068 -                  end
1069 -                  block_temp(group_i).total=block_temp(group_i).total-j;
1070 -      end
1071 -  end
1072 -  group=[];
1073 -  group=group1;%groupの情報の更新
1074
1075    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%新たノードの番号を確認, 例えば(5,2,4,1,3)----->(1,5,2,4,3)
```

図 6.53: block_construction.m（２２）

```
1076 -   box=[];
1077 -   for group_i=1:total_group;
1078 -       if search(group_i)==OK;
1079 -           box_i=0;
1080 -           for task_i=1:taskgraph.total_task+2;
1081 -               check(task_i)=0;
1082 -           end
1083 -           for i=1:block_temp(group_i).total;
1084 -               check(group(group_i,i))=1;%group内のノードを色をつけ
1085 -           end
1086 -           for i=1:block_temp(group_i).total;
1087 -               for ip_i=1:taskgraph.task(group(group_i,i)).ip;
1088 -                   check_flag=1;
1089 -                   ipnumber_i=taskgraph.task(group(group_i,i)).ip_number(ip_i);
1090 -                   if check(ipnumber_i)==1;%調べるノードがgroup中で一番上のノードではないときに
1091 -                       check_flag=0;
1092 -                       break;
1093 -                   end
1094 -               end
1095
1096 -               if check_flag==1;
1097 -                   box_i=box_i+1;
1098 -                   box(group_i,box_i)=group(group_i,i);%新たなノード番号を確定
1099 -                   break;
1100 -               end
1101 -           end
1102 -           for i=1:block_temp(group_i).total;
1103 -               if box(group_i,1)~=group(group_i,i);%残したノードがそのまま代入
1104 -                   box_i=box_i+1;
1105 -                   box(group_i,box_i)=group(group_i,i);
1106 -               end
1107 -           end
1108
1109 -       end
1110 -   end
1111     %box
1112
1113 -   group=box;%group2度目更新
1114     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
1115     %各ブロック完成
1116 -   block_number=[];
1117 -   for group_i=1:total_group;
1118 -       if search(group_i)==OK;%条件を満たすパターンはブロックになる
1119 -           total_block=total_block+1;%(総ブロック数を格納)
1120 -           block_number(total_block)=block_temp(group_i).total;%(メンバの数を格納)
1121 -           for block_i=1:block_temp(group_i).total;
1122 -               block(total_block,block_i)=group(group_i,block_i);%(メンバの番号を格納)
1123 -           end
1124 -       end
1125 -   end
```

図 6.54: block_construction.m（２３）

```
1126      %block_number;
1127      %total_block;
1128
1129 -    fprintf('<<追加ブロック化法で>>で得られたブロック：¥n')
1130 -    disp(block);
1131 -    fprintf('現在のタスクグラフのブロック総数： (%d) ¥n',total_block)
1132      %block_number(3)
1133
1134
1135
1136
1137      %redundant開始
1138
1139 -    for block_i=1:total_block;
1140          %disp(block(block_i,1));
1141 -        redundant(block_i)=0;
1142 -        flag_exit=0;
1143 -        for task_i=1:taskgraph.total_task+2;
1144 -            check(task_i)=0;
1145 -        end
1146 -        for i=1:block_number(block_i);
1147 -            check(block(block_i,i))=1;
1148 -        end
1149 -            for i=1:block_number(block_i);%ブロック中の各nodeたち
1150 -                for is_i=1:taskgraph.task(block(block_i,i)).is%先祖たちの各子たち
1151 -                    isnumber_i=taskgraph.task(block(block_i,i)).is_number(is_i);
1152 -                            if check(isnumber_i)==0;%子が色付いていない
1153 -                                for suc_i=1:taskgraph.task(isnumber_i).suc;%子の子孫確認
1154 -                                    if check(taskgraph.task(isnumber_i).suc_number(suc_i))==1;%もし子孫ただ一人が色付けたら
1155 -                                        flag_exit=1;
1156 -                                        break;
1157 -                                    end
1158 -                                end
1159 -                            end
1160 -                            if flag_exit==1;
1161 -                                break;
1162 -                            end
1163 -                end
1164 -                if flag_exit==1;
1165 -                    disp('ブロック化できないブロックの確認');
1166 -                    disp(block(block_i,:));%だめなメンバたち
1167 -                    redundant(block_i)=1;
1168 -                   break;
1169 -                end
1170 -            end
1171 -    end
1172      %redundant終了
1173
1174
1175 -    for block_i=1:total_block;
```

図 6.55: block_construction.m（２４）

```
1176        %disp(block(block_i,1));
1177 -      if redundant(block_i)==0;
1178
1179 -  chai=zeros(1,2);%最後の対応図のために長いブロックがーーー＞２つタスクを持つグループに変更
1180 -  chai(1,1)=block(block_i,1);
1181 -  for i=2:block_number(block_i);
1182        %block---->group(拆)
1183 -      chai(1,2)=block(block_i,i);
1184 -      groupoftaskgraph=[groupoftaskgraph;chai];
1185 -      chai(1,2)=0;
1186 -  end
1187
1188 -  fprintf('新たなブロックの番号は(  %d  ).¥n', block(block_i,1));
1189 -  fprintf('そのブロックに含まれるノード:¥n');
1190 -  for i=1:block_number(block_i);
1191 -      fprintf('%d ', block(block_i,i));
1192 -  end
1193 -  fprintf('¥n')
1194 -  fprintf('¥n')
1195 -  fprintf('☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆¥n')
1196
1197    %disp(block(block_i,1));
1198    %%%%%%%%%%%%%%%%ブロックが新たなノードに変換し，その親の情報<2>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
1199 -  for task_i=1:taskgraph.total_task+2;%すべてのノードが初期化
1200 -      condition(task_i)=NG;
1201 -  end
1202    %block循環
1203 -  condition(block(block_i,1))=OK;
1204 -  for ip_i=1:taskgraph.task(block(block_i,1)).ip;
1205 -      condition(taskgraph.task(block(block_i,1)).ip_number(ip_i))=OK;
1206 -  end
1207
1208 -  for i=2:block_number(block_i);
1209 -      taskgraph.task(block(block_i,1)).exetime=taskgraph.task(block(block_i,1)).exetime+taskgraph.task(block(block_i,i)).exetime;
1210 -      condition(block(block_i,i))=OK;
1211 -  end
1212
1213 -  for i=2:block_number(block_i);
1214 -      for ip_i_temp=1:taskgraph.task(block(block_i,i)).ip;
1215 -          if condition(taskgraph.task(block(block_i,i)).ip_number(ip_i_temp))~=OK;
1216 -              ip_i=ip_i+1;
1217 -              taskgraph.task(block(block_i,1)).ip_number(ip_i)=taskgraph.task(block(block_i,i)).ip_number(ip_i_temp);
1218 -              taskgraph.task(block(block_i,1)).ip_read(ip_i)=taskgraph.task(block(block_i,i)).ip_read(ip_i_temp);
1219 -              taskgraph.task(block(block_i,1)).ip=taskgraph.task(block(block_i,1)).ip+1;
1220 -              condition(taskgraph.task(block(block_i,i)).ip_number(ip_i_temp))=OK;
1221 -          end
1222 -      end
1223 -  end
1224
1225 -  for task_i=1:taskgraph.total_task+2;%すべてのノードが初期化
```

図 6.56: block_construction.m（２５）

```
1226 -         condition(task_i)=NG;
1227 -     end
1228      %block循環(完)
1229      %%%%%%%%%%%%%%%%%%%%すべてのノードの親の情報の更新<5>%%%%%%%%%%%%%%%%%%%%
1230 -   for task_i=1:taskgraph.total_task+2;
1231 -       for ip_i=1:taskgraph.task(task_i).ip;
1232 -           for i=2:block_number(block_i);
1233 -               if taskgraph.task(task_i).ip_number(ip_i)==block(block_i,i);
1234 -                  taskgraph.task(task_i).ip_number(ip_i)=block(block_i,1);
1235 -               end
1236 -           end
1237 -       end
1238 -   end
1239      %%%%%%%%%%%%%%%%%%%%%%%%要らないノードの情報を捨てる<7>%%%%%%%%%%%%%%%%%%%%%
1240 -   for i=2:block_number(block_i);
1241 -           taskgraph.task(block(block_i,i)).exetime=0;
1242 -           taskgraph.task(block(block_i,i)).ip=0;
1243 -           taskgraph.task(block(block_i,i)).ip_number=0;
1244 -           taskgraph.task(block(block_i,i)).ip_read=0;
1245 -           taskgraph.task(block(block_i,i)).is=0;
1246 -           taskgraph.task(block(block_i,i)).sum_write_time=0;
1247 -           taskgraph.task(block(block_i,i)).sum_read_time=0;
1248 -           taskgraph.task(block(block_i,i)).is_number=0;
1249 -           taskgraph.task(block(block_i,i)).pre=0;
1250 -           taskgraph.task(block(block_i,i)).pre_number=0;
1251 -           taskgraph.task(block(block_i,i)).suc=0;
1252 -           taskgraph.task(block(block_i,i)).suc_number=0;
1253 -           taskgraph.task(block(block_i,i)).pre_number_temp=0;
1254 -           taskgraph.task(block(block_i,i)).suc_number_temp=0;
1255 -   end
1256      %disp('確認');
1257      %disp(taskgraph.task(block(block_i,1)).ip_number);
1258      %disp(taskgraph.task(block(block_i,1)).ip_read);
1259      %%%%%%%%%%%%%%%%%%%%%%%%%親のデータの並び（昇）<8>%%%%%%%%%%%%%%%%%%%%%(通信時間と親がsort)
1260 -   for task_i=1:taskgraph.total_task+2;
1261 -       for ip_i=1:taskgraph.task(task_i).ip;
1262 -           min_no=ip_i;
1263 -           min_number=taskgraph.task(task_i).ip_number(ip_i);
1264 -           min_read=taskgraph.task(task_i).ip_read(ip_i);
1265 -           for ip_i_temp=ip_i+1:taskgraph.task(task_i).ip;
1266 -               if taskgraph.task(task_i).ip_number(ip_i_temp)<min_number;
1267 -                   min_no=ip_i_temp;
1268 -                   min_number=taskgraph.task(task_i).ip_number(ip_i_temp);
1269 -                   min_read=taskgraph.task(task_i).ip_read(ip_i_temp);
1270 -               end
1271 -           end
1272 -           taskgraph.task(task_i).ip_number(min_no)=taskgraph.task(task_i).ip_number(ip_i);
1273 -           taskgraph.task(task_i).ip_read(min_no)=taskgraph.task(task_i).ip_read(ip_i);
1274 -           taskgraph.task(task_i).ip_number(ip_i)=min_number;
1275 -           taskgraph.task(task_i).ip_read(ip_i)=min_read;
```

図 6.57: block_construction.m（２６）

```
1276 -         end
1277 -     end
1278
1279      %disp(taskgraph.task(block(block_i,1)).ip_number);
1280      %disp(taskgraph.task(block(block_i,1)).ip_read);
1281      %%%%%%%%%%%%%%%%%%%%%%%%%同じ結果を削除(親)<10>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
1282 -    for task_i=2:taskgraph.total_task+2;
1283 -        if taskgraph.task(task_i).ip~=0;%情報なしのノード情報が実行しない
1284 -        p=taskgraph.task(task_i).ip_number(1);
1285 -        task_sample(task_i).ip_number_temp(1)=taskgraph.task(task_i).ip_number(1);
1286 -        task_sample(task_i).ip_read_temp(1)=taskgraph.task(task_i).ip_read(1);
1287 -        j=0;%削除した数
1288 -        for i=2:taskgraph.task(task_i).ip;
1289 -            if p==taskgraph.task(task_i).ip_number(i);
1290 -                j=j+1;
1291 -            else
1292 -                task_sample(task_i).ip_number_temp(i-j)=taskgraph.task(task_i).ip_number(i);
1293 -                task_sample(task_i).ip_read_temp(i-j)=taskgraph.task(task_i).ip_read(i);
1294 -                p=taskgraph.task(task_i).ip_number(i);
1295 -            end
1296 -        end
1297 -        taskgraph.task(task_i).ip=taskgraph.task(task_i).ip-j;
1298 -        end%if文のendだ！，情報なしノードがこのプログラムを参加さいない
1299 -    end
1300 -    for task_i=2:taskgraph.total_task+2;
1301 -        taskgraph.task(task_i).ip_number=task_sample(task_i).ip_number_temp;
1302 -        taskgraph.task(task_i).ip_read=task_sample(task_i).ip_read_temp;
1303 -    end
1304      %%%%%%%%%%%%%%%%%%%%%ブロック後のtaskgraphデータを整理する<11>%%%%%%%%%%%%%%%%
1305 -    graph=[];
1306      %graph(1,1)=taskgraph.total_task;
1307 -    for graph_i=1:taskgraph.total_task+2;%sとtとgraph(1,1)
1308 -        graph(graph_i,1)=taskgraph.task(graph_i).task_number;
1309 -        graph(graph_i,2)=taskgraph.task(graph_i).exetime;
1310 -        graph(graph_i,3)=taskgraph.task(graph_i).ip;
1311 -        graph_j=4;
1312 -        for ip_i=1:taskgraph.task(graph_i).ip;
1313 -            graph(graph_i,graph_j)=taskgraph.task(graph_i).ip_number(ip_i);
1314 -            graph_j=graph_j+1;
1315 -            graph(graph_i,graph_j)=taskgraph.task(graph_i).ip_read(ip_i);
1316 -            graph_j=graph_j+1;
1317 -        end
1318 -    end
1319      %graph
1320 -    break;%ブロックらが１つを実行だけ.
1321 -        end%redundant(完)
1322 -        if block_i==total_block;%(プログラム終了条件)
1323 -            exit_program=1;
1324 -        end
1325 -    end%block循環(完)
```

図 6.58: block_construction.m（２７）

```
1326
1327     %%%%%%%%%%%%%%%%%%%%%graphの出力1%%%%%%%%%%%%%%%%
1328 -   graph_out=[];
1329 -   graph_out(1,1)=taskgraph.total_task;
1330 -   for graph_i=2:taskgraph.total_task+3;%sとtとgraph(1,1)
1331 -       graph_out(graph_i,1)=taskgraph.task(graph_i-1).task_number;
1332 -       graph_out(graph_i,2)=taskgraph.task(graph_i-1).exetime;
1333 -       graph_out(graph_i,3)=taskgraph.task(graph_i-1).ip;
1334 -       graph_j=4;
1335 -       for ip_i=1:taskgraph.task(graph_i-1).ip;
1336 -           graph_out(graph_i,graph_j)=taskgraph.task(graph_i-1).ip_number(ip_i);
1337 -           graph_j=graph_j+1;
1338 -           graph_out(graph_i,graph_j)=taskgraph.task(graph_i-1).ip_read(ip_i);
1339 -           graph_j=graph_j+1;
1340 -       end
1341 -   end
1342     %graph
1343     %%%%%%%%%%%%%%%%%%%%%graphの出力2%%%%%%%%%%%%%%%%
1344 -   for task_i=1:taskgraph.total_task+2;
1345 -       checkzero(task_i)=0;
1346 -   end
1347 -   checkzero(1)=1;
1348 -   newtotal=0;
1349
1350 -   for task_i=2:taskgraph.total_task+2;
1351 -       if taskgraph.task(task_i).ip~=0;
1352 -           newtotal=newtotal+1;
1353 -          checkzero(task_i)=1;
1354 -       end
1355 -   end
1356 -   newtotal=newtotal-1;%tノードを除く
1357     %初期化終了
1358 -   graph_out1=[];
1359 -   graph_out1(1,1)=newtotal;
1360 -   graph_ii=2;
1361 -   for graph_i=2:taskgraph.total_task+3;%sとtとgraph(1,1)
1362 -       if checkzero(graph_i-1)==1;
1363 -       graph_out1(graph_ii,1)=taskgraph.task(graph_i-1).task_number;
1364 -       graph_out1(graph_ii,2)=taskgraph.task(graph_i-1).exetime;
1365 -       graph_out1(graph_ii,3)=taskgraph.task(graph_i-1).ip;
1366 -       graph_j=4;
1367 -       for ip_i=1:taskgraph.task(graph_i-1).ip;
1368 -           graph_out1(graph_ii,graph_j)=taskgraph.task(graph_i-1).ip_number(ip_i);
1369 -           graph_j=graph_j+1;
1370 -           graph_out1(graph_ii,graph_j)=taskgraph.task(graph_i-1).ip_read(ip_i);
1371 -           graph_j=graph_j+1;
1372 -       end
1373 -       graph_ii=graph_ii+1;
1374 -       end
1375 -   end
```

図 6.59: block_construction.m（２８）

```
1376
1377
1378     %出力ファイル1を作る**********************************************************
1379 -   fid=fopen('Result_Graph.txt','w');
1380 -   for x=1:taskgraph.total_task+3;%循環=for
1381 -       y=[graph_out(x,:)];
1382 -       z=y';%yのデータの (行->列) 変換,結果zに代入
1383 -           fprintf(fid,'%d ',z);%一行の内容が表示できる
1384 -           fprintf(fid,' ¥n',z);%最後に新たな行に変更
1385 -   end
1386     %出力ファイル2を作る(ノードの情報がない場合表示しない)**********************************************************
1387 -   fid1=fopen('New_Graph.txt','w');
1388 -   for x=1:newtotal+3;%循環=for
1389 -       y=[graph_out1(x,:)];
1390 -       z=y';%yのデータの (行->列) 変換,結果zに代入
1391 -           fprintf(fid1,'%d ',z);%一行の内容が表示できる
1392 -           fprintf(fid1,' ¥n',z);%最後に新たな行に変更
1393 -   end
1394
1395 -   end
1396 -   end
1397
1398
1399
1400     %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
1401     %<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<先祖>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
1402     function Predecessor
1403     %*************************<<先祖を求めるプログラム>>*************************
1404     %(8)各ノードの先祖を求める
1405 -   OK=1;%初期化
1406 -   NG=0;
1407 -   for condition_i=1:taskgraph.total_task+2;
1408 -       condition(condition_i)=NG;
1409 -       taskgraph.task(condition_i).pre=0;
1410 -   end
1411 -   condition(1)=OK;%ｓノードを実行完了
1412 -   end_task=1;%実行完了の数が１となる
1413 -   end_temp(1)=1;%実行完了のノード番号
1414 -   for task_i=1:taskgraph.total_task+2;%先祖の配列中でまず各ノードの親の情報を代入
1415 -       taskgraph.task(task_i).pre=taskgraph.task(task_i).ip;
1416 -       task_sample(task_i).pre_number=zeros(1,taskgraph.total_task+1);
1417         %臨時task_sample構造体を作る．行のサイズが'ｔ'ノードを除いてタスクグラフの中のノードの総数
1418 -       for ip_i=1:taskgraph.task(task_i).ip;
1419 -           task_sample(task_i).pre_number(ip_i)=taskgraph.task(task_i).ip_number(ip_i);%各ノード先祖の情報はとりあえず臨時構造体に保存
1420 -       end
1421 -   end
1422     %プログラムを実行開始
1423 -   while(end_task<taskgraph.total_task+2);
1424 -       for task_i=1:taskgraph.total_task+2;
1425 -           if condition(task_i)==NG;
```

図 6.60: block_construction.m（２９）

```
1426 -                  condition(task_i)=OK;
1427 -                  for ip_i=1:taskgraph.task(task_i).ip;%親ノードが実行したどうか確認する
1428 -                      check=NG;
1429 -                      for end_i=1:end_task;
1430 -                          if task_sample(task_i).pre_number(ip_i)==end_temp(end_i);
1431 -                              check=OK;
1432 -                              break;
1433 -                          end
1434 -                      end
1435 -                      if check==NG;
1436 -                          condition(task_i)=NG;
1437 -                          break;
1438 -                      end
1439 -                  end
1440      %もし条件を満たしならば．(親と子同じ先祖を持ってる場合，データの代入などを実行)
1441 -     if condition(task_i)==OK;
1442 -         for ip_i=1:taskgraph.task(task_i).ip;
1443 -             ip_number_temp=taskgraph.task(task_i).ip_number(ip_i);
1444 -             for ip_pre_i=1:taskgraph.task(ip_number_temp).pre;
1445 -                 for pre_i=1:taskgraph.task(task_i).pre;
1446 -                     check=OK;
1447 -                     if task_sample(task_i).pre_number(pre_i)==task_sample(ip_number_temp).pre_number(ip_pre_i);
1448 -                         check=NG;
1449 -                         break;
1450 -                     end
1451 -                 end
1452 -                 if check==OK;
1453 -                     task_sample(task_i).pre_number(pre_i+1)=task_sample(ip_number_temp).pre_number(ip_pre_i);
1454                      %(pre_i+1)ということは「c言語のforは最後i++」「matlabのfor最後i」
1455 -                     taskgraph.task(task_i).pre=taskgraph.task(task_i).pre+1;
1456 -                 end
1457 -             end
1458 -         end
1459 -         end_task=end_task+1;
1460 -         end_temp(end_task)=task_i;
1461 -     end
1462 -         end
1463 -     end
1464 - end
1465   %ちょうど大きさ先祖配列を用意する
1466 - for i=1:taskgraph.total_task+2;
1467 -         taskgraph.task(i).pre_number=zeros(1,taskgraph.task(i).pre);
1468 - end
1469   %臨時構造体の先祖の情報をタスクグラフの構造体に代入して，sortする
1470 - for i=1:taskgraph.total_task+2;
1471 -     for j=1:taskgraph.task(i).pre;
1472 -         taskgraph.task(i).pre_number(j)=task_sample(i).pre_number(j);%データの代入
1473 -     end
1474 -     taskgraph.task(i).pre_number=sort(taskgraph.task(i).pre_number);%昇順並び
1475   %disp(taskgraph.task(i));%データの確認
```

図 6.61: block_construction.m（３０）

```
1476 -  end
1477 -  end
1478
1479    %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
1480    %<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<子孫>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
1481    function Successor
1482    %**************************<<子孫を求めるプログラム>>**************************
1483    %(9)各ノードの子孫を求める
1484 -  OK=1;%初期化
1485 -  NG=0;
1486 -  for condition_i=1:taskgraph.total_task+2;
1487 -      condition(condition_i)=NG;
1488 -      taskgraph.task(condition_i).suc=0;
1489 -  end
1490 -  condition(taskgraph.total_task+2)=OK;%tノードを実行完了
1491 -  end_task=1;%実行完了の数が１となる
1492 -  end_temp(1)=taskgraph.total_task+2;%実行完了のノード番号
1493 -  for task_i=1:taskgraph.total_task+2;%子孫の配列中でまず各ノードの親の情報を代入
1494 -      taskgraph.task(task_i).suc=taskgraph.task(task_i).is;
1495 -      task_sample(task_i).suc_number=zeros(1,taskgraph.total_task+1);
1496        %臨時task_sample構造体を作る．行のサイズが’ｔ’ノードを除いてタスクグラフの中のノードの総数
1497 -      for is_i=1:taskgraph.task(task_i).is;
1498 -          task_sample(task_i).suc_number(is_i)=taskgraph.task(task_i).is_number(is_i);%各ノード子孫の情報はとりあえず臨時構造体に保存
1499 -      end
1500 -  end
1501    %プログラムを実行開始
1502 -  while(end_task<taskgraph.total_task+2);
1503 -      for task_i=1:taskgraph.total_task+2;
1504 -          if condition(task_i)==NG;
1505 -              condition(task_i)=OK;
1506 -              for is_i=1:taskgraph.task(task_i).is;%子ノードが実行したどうか確認する
1507 -                  check=NG;
1508 -                  for end_i=1:end_task;
1509 -                      if task_sample(task_i).suc_number(is_i)==end_temp(end_i);
1510 -                          check=OK;
1511 -                          break;
1512 -                      end
1513 -                  end
1514 -                  if check==NG;
1515 -                      condition(task_i)=NG;
1516 -                      break;
1517 -                  end
1518 -              end
1519     %もし条件を満たしならば．(子と親同じ子孫を持ってる場合，データの代入などを実行)
1520 -   if condition(task_i)==OK;
1521 -       for is_i=1:taskgraph.task(task_i).is;
1522 -           is_number_temp=taskgraph.task(task_i).is_number(is_i);
1523 -           for is_suc_i=1:taskgraph.task(is_number_temp).suc;
1524 -               for suc_i=1:taskgraph.task(task_i).suc;
1525 -                   check=OK;
```

図 6.62: block_construction.m（３１）

```
1526 -                  if task_sample(task_i).suc_number(suc_i)==task_sample(is_number_temp).suc_number(is_suc_i);
1527 -                      check=NG;
1528 -                      break;
1529 -                  end
1530 -              end
1531 -              if check==OK;
1532 -                  task_sample(task_i).suc_number(suc_i+1)=task_sample(is_number_temp).suc_number(is_suc_i);
1533                   %(suc_i+1)ということは「c言語のforは最後i++」「matlabのfor最後i」
1534 -                  taskgraph.task(task_i).suc=taskgraph.task(task_i).suc+1;
1535 -              end
1536 -          end
1537 -      end
1538 -      end_task=end_task+1;
1539 -      end_temp(end_task)=task_i;
1540 -  end
1541 -          end
1542 -      end
1543 - end
1544   %ちょうど大きさ子孫配列を用意する
1545 - for i=1:taskgraph.total_task+2;
1546 -         taskgraph.task(i).suc_number=zeros(1,taskgraph.task(i).suc);
1547 - end
1548   %臨時構造体の子孫の情報をタスクグラフの構造体に代入して，sortする
1549 - for i=1:taskgraph.total_task+2;
1550 -     for j=1:taskgraph.task(i).suc;
1551 -         taskgraph.task(i).suc_number(j)=task_sample(i).suc_number(j);%データの代入
1552 -     end
1553 -     taskgraph.task(i).suc_number=sort(taskgraph.task(i).suc_number);%昇順並び
1554   %disp(taskgraph.task(i));%データの確認
1555 - end
1556 - end
1557
1558
1559   %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
1560   %*************************<<対応図を作るプログラム>>*************************
1561   function blocktable
1562 - group=groupoftaskgraph;
1563   %disp(group);%読み込みデータの表示
1564 - array_size=size(group);%サイズを求め，結果はarray_sizeに代入
1565 - total_group=array_size(1);%列数をgroupの総数として使う
1566   %%%%%%%%%%%%%%%%%%%%%%%%%%%%各パターンの組み合わせ%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
1567 - Complete=1;
1568 - InComplete=0;
1569 - OK=1;
1570 - NG=0;
1571 - total_block=0;%ブロックの数の初期化
1572
1573 - for task_i=1:taskgraph.total_task+2;
1574 -     check(task_i)=0;%ブロック含めるかどうかを判断する
1575 - end
```

図 6.63: block_construction.m（３２）

```
1576
1577 -  for group_i=1:total_group;%[[[[[すべてgrouoの中身]]]]
1578 -      block_temp(group_i).total=2;%各ブロックの中身の数の初期化
1579 -      condition(group_i)=InComplete;%各パターンが実行したかどうか
1580 -      search(group_i)=NG;%パターンがブロックに変更する条件
1581 -  end
1582
1583
1584 -  for group_i=1:total_group;
1585 -          for group_i_temp=group_i+1:total_group;
1586 -                  if group(group_i,1)==group(group_i_temp,1)&&group(group_i,2)==group(group_i_temp,2)
1587 -                      condition(group_i_temp)=Complete;
1588 -                  end
1589 -          end
1590 -  end
1591
1592 -  for group_i=1:total_group;
1593 -      if condition(group_i)==InComplete;
1594 -          while (1);
1595 -               finish(group_i)=block_temp(group_i).total;
1596 -          for group_i_temp=group_i+1:total_group;
1597 -              if condition(group_i_temp)==InComplete;
1598                   %if group_i~=group_i_temp;
1599 -                     for i=1:block_temp(group_i).total;
1600 -                         if group(group_i,i)==group(group_i_temp,1);
1601 -                          group(group_i,block_temp(group_i).total+1)=group(group_i_temp,2);%パターンに新たな内容を追加する
1602 -                           block_temp(group_i).total=block_temp(group_i).total+1;
1603 -                           condition(group_i_temp)=Complete;
1604 -                         end
1605 -                          if group(group_i,i)==group(group_i_temp,2);
1606 -                              group(group_i,block_temp(group_i).total+1)=group(group_i_temp,1);%パターンに新たな内容を追加する
1607 -                              block_temp(group_i).total=block_temp(group_i).total+1;
1608 -                              condition(group_i_temp)=Complete;
1609 -                         end
1610 -                     end
1611                   %end
1612 -              end
1613 -          end
1614 -          if finish(group_i)==block_temp(group_i).total;
1615           %disp('確認');
1616 -          break;
1617 -          end
1618 -          end
1619 -          condition(group_i)=Complete;
1620 -          search(group_i)=OK;
1621 -      end
1622 -  end
1623 -  for group_i=1:total_group;
1624 -      if search(group_i)==OK;%条件を満たすパターンはブロックになる
1625 -          total_block=total_block+1;
```

図 6.64: block_construction.m（３３）

```
1626 -         for block_i=1:block_temp(group_i).total;
1627 -             block(total_block,block_i)=group(group_i,block_i);
1628 -             check(group(group_i,block_i))=1;%色付け
1629 -         end
1630 -         blocksize(total_block)=block_temp(group_i).total;
1631 -     end
1632 - end
1633
1634 - blocksize;
1635 - table=[];
1636 - for block_i=1:total_block;
1637 -     for size_i=1:blocksize(block_i);
1638 -         table(block_i,size_i+1)=block(block_i,size_i);
1639 -     end
1640 - end
1641   %total_block
1642   %disp(block);
1643 - j=0;
1644 - for task_i=1:taskgraph.total_task+2;
1645 -     if taskgraph.task(task_i).ip>0&&check(task_i)==1;
1646 -        j=j+1;
1647 -        a(j,1)=task_i;
1648 -     end
1649 - end
1650   %a
1651 - for a_i=1:j;
1652 - for block_i=1:total_block;
1653 -     for size_i=2:blocksize(block_i)+1;
1654 -         if a(a_i)==table(block_i,size_i);
1655 -             table(block_i,1)=a(a_i);
1656 -             break;
1657 -         end
1658 -     end
1659 - end
1660 - end
1661   %table
1662   %すべての結果を提出する
1663 - fid2=fopen('Node_Table.txt','w');
1664 - for x2=1:total_block;
1665 -     y2=[table(x2,:)];
1666 -     z2=y2';
1667 -     fprintf(fid2,'%d ',z2);
1668 -     fprintf(fid2,' ¥n',z2);
1669 - end
1670
1671
1672 - Box=[1,];
1673   %☆☆☆☆☆☆☆☆☆☆☆☆☆☆☆最後結果を表示☆☆☆☆☆☆☆☆☆☆☆☆☆☆
1674 - fprintf('ブロック総数は %d  個.¥n', total_block)
1675 - fprintf('それぞれのブロックが新たなノードに変更し¥n')
```

図 6.65: block_construction.m（３４）

```
1676 -   fprintf('¥n')
1677 -   fprintf('¥n')
1678 -  for block_i=1:total_block;
1679 -   fprintf('ブロック<%d>から変更したノード番号は %d です.¥n', block_i, table(block_i,1))
1680 -   fprintf('対応するノード番号は：¥n')
1681 -  for size_i=1:blocksize(block_i);
1682 -           Box(size_i)=block(block_i,size_i);
1683 -   end
1684 -   Box=sort(Box);
1685 -   disp(Box);
1686 -   Box=[1,];
1687 -   end
1688
1689
1690 -   end
1691
1692
1693
1694     %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
1695    function check
1696 -   disp('親の数が0となるタスク：');
1697 -   i=0;
1698 -   j=0;
1699     %ip=[];
1700     %is=[];
1701 -  for task_i=1:taskgraph.total_task+2;
1702 -       if taskgraph.task(task_i).ip==0;
1703 -           i=i+1;
1704 -           ip(i)=task_i;
1705 -           disp(task_i);
1706 -       end
1707 -   end
1708 -   i
1709 -   disp('子の数が0となるタスク：');
1710 -  for task_i=1:taskgraph.total_task+2;
1711 -       if taskgraph.task(task_i).is==0;
1712 -           j=j+1;
1713 -           is(j)=task_i;
1714 -           disp(task_i);
1715 -       end
1716 -   end
1717 -   j
1718 -   end
1719
1720
1721 -   end
```

図 6.66: block_construction.m（３５）

(3) 次は再ブロック化法のプログラムを示す．本プログラムの実行ファイルは block_reconstruction.m という名前のスクリプト M-ファイルである．このプログラムは 4 つの関数が存在する．(i)「Chen」はタスクグラフの情報をファイルから読み込み，タスクグラフまた各ノードの構造体を構成する関数である．(ii)「List」はデットラインを用いたスケジューリング法に基づいてタスクグラフの優先リストを求める関数である．(iii)「Scheduling」は再ブロック化法専用のスケジューリング法 (LST-EST 法) を利用して，タスクグラフのスケジュール結果を求める関数である．(iv)「ReBlock」は提案した再ブロック化法を用いてタスクグラフ中のノードをブロックする関数である．なお本プログラムはブロックできる限り循環に実行することである．

```
1     function block_reconstruction
2     %*************<<タスクグラフの各タスクの基本情報を構造体へ>>****************
3     %(1)ファイルの読み込み (開始)
4  -  taskgraph_name=input('taskgraphの名前を入力してください：','s');
5  -  fd=fopen(taskgraph_name,'r');%ファイルを開く
6  -  line=fgets(fd);%第一行を読み込み (文字列)
7  -  array=str2num(line);% (文字列->数字) 関数を使う
8  -  number_total=array(1);%配列の第一の変数を使う
9  -  datas=[];%空の配列を用意
10 -  for line_m=1:number_total+2;%ファイル中のすべてのデータを読み込み
11 -      line=fgets(fd);
12 -  temp=str2num(line);
13 -  datas=[datas;temp];%配列の追加
14 -  end
15    %disp(datas);%配列の表示*****************************************************
16 -  fclose(fd);%ファイルの読み込み (終わり)
17 -  groupoftaskgraph=[];
18 -  times=0;
19 -  exit_program=1;
20 -  graph=datas;
21 -  for i=1:2;
22    %while(1)
23 -  task_sample=[];
24 -  taskgraph=[];
25 -  exe_number=[];
```

図 6.67: block_reconstruction.m（1）

```
26 -    range=[];
27 -    range_number=[];
28 -    data=[];
29 -    Chen
30 -    List
31 -    Scheduling
32 -    ReBlock
33
34 -    if program_finish==0;
35 -        disp('★★★★★このタスクグラフにブロック化法できない★★★');
36 -        disp('★★★実行できるブロックの数は"0"ので，プログラムそのまま終了★★★');
37 -        break;
38 -    end
39 -    times=times+1;
40 -   end
41 -   fprintf('再ブロック化法の実行回数：( %d ) 回.¥n', times);
42
43      %<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<入力>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
44     function Chen
45 -       taskgraph.total_task=number_total;%タスクの総数を求める（構造体に追加）
46         %number_total
47 -       array_size=zeros(1,2);%配列のサイズの配列を用意
48 -       array_size=size(graph);%サイズのデータの代入
49         %disp(array_size(1));disp(array_size(2));
50     %(2)タスクグラフの情報を表示（構造体）
51 -   for i=1:number_total+2;%配列中のデーターを一行一行で代入
52 -       line_x=1;%第2列（ノードの実行時間）
53 -       read_i=4;%第4列から（ノードの各親の番号と通信時間の情報）
54 -       ipread_i2=0;%ノードの各親の番号を求めるために
55 -       ipread_i3=0;%ノードの各通信時間を求めるために
56 -       for array_m=1:array_size(2);%データー配列の列の数
57 -           if line_x==1;
58 -               taskgraph.task(i).task_number=graph(i,array_m);
59 -               line_x=line_x+1;
60 -           elseif line_x==2;%ノードの実行時間を代入
61 -       taskgraph.task(i).exetime=graph(i,array_m);
62 -       line_x=line_x+1;
63 -           elseif line_x==3; %ノードの親の数の情報を代入
64 -       taskgraph.task(i).ip=graph(i,array_m);
65 -       line_x=line_x+1;
66 -           elseif line_x>3;
67 -               if read_i==line_x;%ノードの各親の番号情報を代入
68 -                   ipnumber_i=line_x-3;
69 -                   ipread_i=ipnumber_i-ipread_i2;
70                     %taskgraph.task(i).ip_number(ipread_i)=datas(i,array_m);
71 -                   task_sample(i).ip_number(ipread_i)=graph(i,array_m);
72 -                   line_x=line_x+1;
73 -                   ipread_i2=ipread_i2+1;
74 -               elseif read_i==line_x-1;%ノードの各通信時間の情報を代入
75 -                   ipnumber_i=line_x-4;
```

図 6.68: block_reconstruction.m（2）

```
76 -                 ipread_i=ipnumber_i-ipread_i3;
77                   %taskgraph.task(i).ip_read(ipread_i)=datas(i,array_m);
78 -                 task_sample(i).ip_read(ipread_i)=graph(i,array_m);
79 -                 line_x=line_x+1;
80 -                 read_i=read_i+2;
81 -                 ipread_i3=ipread_i3+1;
82 -             end
83 -         end
84 -     end
85 - end
86   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%<<<追加>>>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
87 -            for task_i=1:taskgraph.total_task+2;
88 -                for ip_i=1:taskgraph.task(task_i).ip;
89 -                    taskgraph.task(task_i).ip_number(ip_i)=task_sample(task_i).ip_number(ip_i);
90 -                    taskgraph.task(task_i).ip_read(ip_i)=task_sample(task_i).ip_read(ip_i);
91 -                end
92 -            end
93   %(3)タスクの子供の情報&&読み込み時間&&書き出す時間を求める (構造体情報の追加)
94 - for task_i=1:taskgraph.total_task+2;%各タスクの子と書き出し時間の初期化
95 -     taskgraph.task(task_i).is=0;
96 -     taskgraph.task(task_i).sum_write_time=0;
97 -     taskgraph.task(task_i).sum_read_time=0;
98 - end
99 - taskgraph.total_is=0;%タスクグラフの各タスクの親と子 (初期化)
100 - taskgraph.total_ip=0;
101
102 - for task_i=2:taskgraph.total_task+2;%各タスクの子の情報を追加 (構造体へ)
103 -     for ip_i=1:taskgraph.task(task_i).ip;
104 -         ipnumber_i=taskgraph.task(task_i).ip_number(ip_i);
105 -         is_i=taskgraph.task(ipnumber_i).is+1;
106 -         taskgraph.task(ipnumber_i).is_number(is_i)=task_i;
107 -         taskgraph.task(ipnumber_i).is=taskgraph.task(ipnumber_i).is+1;
108 -     end
109 - end
110 - for task_i=1:taskgraph.total_task+2;%タスクグラフの各タスクの親と子の総数
111 -     taskgraph.total_is=taskgraph.total_is+taskgraph.task(task_i).is;
112 -     taskgraph.total_ip=taskgraph.total_ip+taskgraph.task(task_i).ip;
113 - end
114 - for task_i=1:taskgraph.total_task+2;%各タスクの書き出し時間を求める (構造体へ)
115 -     for is_i=1:taskgraph.task(task_i).is;
116 -         is_number_temp=taskgraph.task(task_i).is_number(is_i);
117 -         for ip_i=1:taskgraph.task(is_number_temp).ip;
118 -             if taskgraph.task(is_number_temp).ip_number(ip_i)==task_i;
119 -                 taskgraph.task(task_i).sum_write_time=taskgraph.task(task_i).sum_write_time+taskgraph.task(is_number_temp).ip_read(ip_i);
120 -             break
121 -             end
122 -         end
123 -     end
124 - end
125 - for task_i=2:taskgraph.total_task+2;%各タスクの書き込み時間を求める (構造体へ)
```

図 6.69: block_reconstruction.m（3）

```
126 -      for ip_i=1:taskgraph.task(task_i).ip;
127 -          taskgraph.task(task_i).sum_read_time=taskgraph.task(task_i).sum_read_time+taskgraph.task(task_i).ip_read(ip_i);
128 -      end
129 -  end
130
131    %disp('各タスクの情報，総数：');%********************************************
132    %disp(taskgraph.total_task+2);
133    %for i=1:taskgraph.total_task+2;
134    %disp(taskgraph.task(i));
135    %end
136    %disp(taskgraph);%************************************************************
137 -  end
138
139    %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
140
141
142    function List
143    %***********************<<優先リスト（デットライン法）>>**********************
144    %(4)各ノードの最長距離を求める
145 -  task_dis=zeros(1,taskgraph.total_task+2);%total_task+2の長さの１元配列を用意する
146    %disp(taskgraph);
147 -  disp(taskgraph.task(2));
148 -  for task_i=1:taskgraph.total_task+2;%初期化（sノードからの距離&&各ノードのチェック)
149 -      task_dis(task_i)=0;
150 -      taskgraph.task(task_i).earliest=0;%★
151 -      check_task(task_i)=-1;
152 -  end
153 -  check_task(1)=0;%(sノードチェック完了)
154    %%%%%%%%%%%%%%%%%%%(ip,is_flag前もって定義する,[改1])%%%%%%%%%%%%%%%%%%%%%%
155 -  ip_flag=7;%        ||もしtaskの情報は何もない時に定義しなければ,エラーを出る||
156 -  is_flag=7;%        ||「7」という数字は自分が好きだけだ                       ||
157    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
158
159 -  while check_task(taskgraph.total_task+2)==-1;
160 -        for task_i=2:taskgraph.total_task+2;
161 -              if check_task(task_i)==-1;%探索中のノードがチェックしていない場合
162                    %task_i
163 -                  for ip_i=1:taskgraph.task(task_i).ip;%すべての親によってチェックかどうかを確認する
164 -                      ip_flag=0;
165 -                      if check_task(taskgraph.task(task_i).ip_number(ip_i))==-1;%親がチェックしていない場合
166 -                          ip_flag=-1;
167 -                          break;
168 -                      end
169 -                  end
170 -                      if ip_flag==0;%親がチェックした場合
171 -                          Maxdis=0;%最大距離の初期化
172 -                          for ip_i=1:taskgraph.task(task_i).ip;%すべての親によって最大距離を求める
173 -                              if  task_dis(taskgraph.task(task_i).ip_number(ip_i))>Maxdis;
174 -                                  Maxdis=task_dis(taskgraph.task(task_i).ip_number(ip_i));
175 -                              end
```

図 6.70: block_reconstruction.m（４）

```
176 -                         end
177 -                         task_dis(task_i)=Maxdis+taskgraph.task(task_i).sum_read_time+taskgraph.task(task_i).exetime+
178 -                         taskgraph.task(task_i).sum_write_time;
179 -                         check_task(task_i)=0;
180 -                         taskgraph.task(task_i).earliest=Maxdis;
181 -                     end
182 -             end
183 -         end
184 - end
185
186   %disp(task_dis);%各ノードの最長距離を確認************************************
187
188   %(5)各ノードの最遅開始時刻を求める(説明は(4)と同じ，省略)
189 - time=zeros(1,taskgraph.total_task+2);%total_task+2の長さの１元配列を用意する
190 - for task_i=1:taskgraph.total_task+2;%初期化（各ノード最遅開始時刻&&各ノードのチェック)
191 -     time(task_i)=0;
192 -     check_task2(task_i)=-1;
193 - end
194 - check_task2(taskgraph.total_task+2)=0;%tノードチェック完了
195 - time(taskgraph.total_task+2)=task_dis(taskgraph.total_task+2);%tノードの最遅開始時刻
196 - while check_task2(1)==-1;
197 - for task_i=1:taskgraph.total_task+1;
198 -     if check_task2(task_i)==-1;
199 -         for is_i=1:taskgraph.task(task_i).is;
200 -             is_flag=0;
201 -             if check_task2(taskgraph.task(task_i).is_number(is_i))==-1;
202 -                 is_flag=-1;
203 -                 break;
204 -             end
205 -         end
206 -         if is_flag==0;
207 -             Mintime=99999;
208 -             for is_i=1:taskgraph.task(task_i).is;
209 -                 if time(taskgraph.task(task_i).is_number(is_i))<Mintime;
210 -                     Mintime=time(taskgraph.task(task_i).is_number(is_i));
211 -                 end
212 -             end
213 -             time(task_i)=Mintime-taskgraph.task(task_i).sum_read_time-taskgraph.task(task_i).exetime-taskgraph.task(task_i).sum_write_time;
214 -             check_task2(task_i)=0;
215 -         end
216 -     end
217 - end
218 - end
219
220   %disp(time);%各ノードの最遅開始時刻を確認
221   %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★(2013追加)（最遅開始時刻の情報が構造体に代入)
222 - for task_i=1:taskgraph.total_task+2;
223 -     taskgraph.task(task_i).latest=time(task_i);
224 -     if taskgraph.task(task_i).latest==99999;%"Mintime"の数字と同じならば(17行前)
225 -         taskgraph.task(task_i).latest=0;
```

図 6.71: block_reconstruction.m（5）

```
226 -            end
227 -        end
228          %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
229          %(6)優先リストを作成
230 -        task_list=zeros(1,taskgraph.total_task+2);%臨時リストを用意（ｓ，ｔを含む）
231          %time=sort(time)
232 -        for task_i=1:taskgraph.total_task+2;
233 -            task_list(task_i)=task_i;
234 -        end
235          %disp(task_list);
236 -        for task_i=2:taskgraph.total_task+1;%ｓ，ｔノードを除く，ノードの昇順に並び
237 -            for task_i_temp=task_i+1:taskgraph.total_task+1;
238 -                if time(task_i)>time(task_i_temp);
239 -                    Min=time(task_i);
240 -                    time(task_i)=time(task_i_temp);
241 -                    time(task_i_temp)=Min;
242 -                    Min=task_list(task_i);
243 -                    task_list(task_i)=task_list(task_i_temp);
244 -                    task_list(task_i_temp)=Min;
245 -                end
246 -            end
247 -        end
248          %disp(task_list);
249 -        taskgraph.priority_list=zeros(1,taskgraph.total_task);%優先リストを用意
250 -        taskgraph.value=zeros(1,taskgraph.total_task);%valueリストを用意
251 -        for task_i=1:taskgraph.total_task;
252 -            taskgraph.priority_list(task_i)=task_list(task_i+1);
253 -            taskgraph.value(task_i)=time(task_i+1);
254 -        end
255          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
256          %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
257          %(7)リスト中の各ノードの範囲を求めるtableを作る
258          %global range;
259          %global range_number;
260          %range=[];%範囲のtableを作る．
261 -        range_number=zeros(1,taskgraph.total_task);
262 -        for task_i=1:taskgraph.total_task;
263 -            range(task_i,1)=taskgraph.priority_list(task_i);
264 -        end
265 -        for range_i=1:taskgraph.total_task;
266 -            count=1;
267 -            for task_i=1:taskgraph.total_task+2;%余計
268 -                if taskgraph.task(task_i).ip~=0;
269 -                if range(range_i,1)==task_i;
270 -                    for list_i=1:taskgraph.total_task;
271 -                        if taskgraph.task(taskgraph.priority_list(list_i)).ip~=0;
272 -                        if taskgraph.task(task_i).latest>=taskgraph.task(taskgraph.priority_list(list_i)).earliest&&
273 -                            taskgraph.task(task_i).latest<=taskgraph.task(taskgraph.priority_list(list_i)).latest;
274 -                            count=count+1;
275 -                            range(range_i,count)=taskgraph.priority_list(list_i);
```

図 6.72: block_reconstruction.m（6）

```
276 -                 end
277 -                 end
278 -             end
279 -             range_number(range_i)=count;
280 -             break;
281 -         end
282 -         end
283 -     end
284 - end
285
286
287
288   %disp(taskgraph.value);
289   %disp(range);
290   %disp(range_number);
291   %disp(taskgraph.priority_list);
292   %for task_i=1:taskgraph.total_task+2;
293   %    disp(taskgraph.task(task_i));
294   %end
295   %disp(taskgraph);
296 - end
297
298   %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
299
300
301
302   function Scheduling
303
304       %**********************<<スケジューリングのプログラム>>**********************
305   %(7)スケジューリングのプログラムを作成
306   %disp(taskgraph);
307   %*****************************<<実行準備>>*********************************
308   %global exe_number
309 - list=taskgraph.priority_list;
310 - PROCESSOR=2;%使用するプロセッサの数（２或いは４）
311 - Complete=1; %（ノードの状態）
312 - InComplete=0; %（ノードの状態）
313 - OK=1; %（チェックの状態）
314 - NG=0; %（チェックの状態）
315 - Free=1; %（プロセッサの状態）
316 - Busy=0; %（プロセッサの状態）
317 - Pbox=zeros(PROCESSOR,2);%(proのboxを用意し，どのプロセッサでどのタスクを実行することを表示)
318   %************************************************************************
319   %初期化（プロセッサの最初状態と各タスクの最初状態)
320 - for pro_i=1:PROCESSOR;
321 -     pro_condition(pro_i)=Free;%プロセッサを使っていない
322 -     end_time(pro_i)=0;%各プロセッサの最短時刻
323 -     Pbox(pro_i,1)=pro_i;
324 - end
325 - for task_i=1:taskgraph.total_task+2;
```

図 6.73: block_reconstruction.m（7）

```
326        %flag2(task_i)=OK;
327 -      check(task_i)=NG;
328 -      condition(task_i)=InComplete;
329 -  end
330 -  condition(1)=Complete;%ｓノードは完成
331 -  total_endtask=0;%チェック完成したノードの数
332 -  flaglost=NG;%リストの指針は最後のデータを指していない場合
333 -  pro_number=PROCESSOR;%残したプロセッサの数を表示
334 -  exe_number=zeros(taskgraph.total_task+2,6);%exe_number配列を用意
335 -  range
336 -  range_number
337    %Pbox
338    %実行開始**********************************************************************
339 -  while total_endtask~=taskgraph.total_task;%総の循環（完成したノード=総ノード数）
340    %for list_i=1:31;
341 -  for list_i=1:taskgraph.total_task;%listの循環
342 -      cannot=1;
343 -      finish=1;
344        %disp('確認');
345        %list(list_i)
346 -      flagend=NG;
347 -      if list(list_i)==list(taskgraph.total_task);%指針がリストの最後データを指したら
348 -          flaglost=OK;
349 -      end
350 -  check1=OK;%ノードの親がチェック完成したどうか
351 -  if condition(list(list_i))==InComplete;%探索開始
352 -      for ip_i=1:taskgraph.task(list(list_i)).ip;%各プロセッサのend_timeの更新（37行~77行）
353 -          ip_number_temp=taskgraph.task(list(list_i)).ip_number(ip_i);
354    %**************「状態１：プロセッサがBusy||Free，実行できるノード完全がない」************
355 -          if condition(ip_number_temp)~=Complete;%親のチェック
356 -              check1=NG;
357 -              if flaglost==OK;%リストの中で選ぶ相手がない時に（リストの最後に指す）
358 -                  Min_temp=9999;
359 -                  for pro_i=1:PROCESSOR;%一番最小のプロセッサ実行値（Min_temp）を確認
360 -                      if pro_condition(pro_i)~=Free;%('使っている'プロセッサだけ選ぶ)
361 -                      if Min_temp>end_time(pro_i);
362 -                          Min_temp=end_time(pro_i);
363 -                      end
364 -                      end
365 -                  end
366 -                  for pro_i=1:PROCESSOR;
367 -                      if Min_temp==end_time(pro_i);%最小値を持っているプロセッサの番号を確認
368 -                          pro_condition(pro_i)=Free;%そのプロセッサが実行完了と設定する．
369 -                          pro_number=pro_number+1;
370 -                          for task_i=1:taskgraph.total_task+2;%実行完了プロセッサがどのタスクを実行する？って確認する
371 -                              if condition(task_i)~=Complete;
372 -                                  if pro_i==exe_number(task_i,2)&&check(task_i)==OK;%taskのデータはexe_numberの配列に入り（OKになる）
373 -                                      condition(task_i)=Complete;%違いプロセッサのデータを比較して，endtime少ない方がチェック完成
374 -                                      flaglost=NG;%実行ノードがありになった
375 -                                      flagend=OK;%優先リストのためにデータ１回実行した後で，優先度高いデータを選ぶ
```

図 6.74: block_reconstruction.m（８）

```
376 -                             end
377 -                         end
378 -                     end
379 -                 end
380 -             end
381 -             for pro_i=1:PROCESSOR;%各プロセッサの最後の時刻が一緒にする（短いプロセッサが長いプロセッサに一致する）
382 -                 if end_time(pro_i)<Min_temp;
383 -                     end_time(pro_i)=Min_temp;
384 -                 end
385 -             end
386 -         end%p42終了
387 -         break;%親の循環に跳び出し
388 -     end
389 - end
390 - if flagend==OK;%リストがもう一度最初から（初期化）
391 -     break;
392 - end
393 %**************「状態2：普通の状態」**********************************
394 - if check1==OK;%親がすべてCompleteならば,データを代入
395 -     for pro_i=1:PROCESSOR;%大循環１
396 -         if pro_condition(pro_i)==Free;%プロセッサがちょうど使っていない,大循環２
397 -             if Pbox(pro_i,2)==0; %if小循環３_1
398
399 -             exe_number(list(list_i),1)=list(list_i);
400 -             exe_number(list(list_i),2)=pro_i;
401 -             exe_number(list(list_i),3)=end_time(pro_i);
402 -             exe_number(list(list_i),4)=taskgraph.task(list(list_i)).sum_read_time+exe_number(list(list_i),3);
403 -             exe_number(list(list_i),5)=taskgraph.task(list(list_i)).exetime+exe_number(list(list_i),4);
404 -             exe_number(list(list_i),6)=taskgraph.task(list(list_i)).sum_write_time+exe_number(list(list_i),5);
405
406 -             end_time(pro_i)=exe_number(list(list_i),6);
407 -             condition(list(list_i))=-1;%タスクが実行中
408 -             check(list(list_i))=OK;%table中に入れた
409 -             Pbox(pro_i,2)=list(list_i);
410 -             cannot=0;
411 -             finish=0;
412 -             total_endtask=total_endtask+1;%完成したタスクが１つ増える
413 -             pro_number=pro_number-1;%使えるプロセッサの数が１つ減らす
414 -             pro_condition(pro_i)=Busy;%そのプロセッサの状態が使えっている状態になった.
415 -             break;%１つのプロセッサが選んだら循環から跳び出し
416 -              end%if小循環３_1
417
418
419 -              if Pbox(pro_i,2)~=0;%if小循環３_2
420 -                 P_temp=Pbox(pro_i,2);%臨時Pbox
421                 %list(list_i)
422                 %range(list_i,1)
423                 %range_number(list_i)
424                 %disp('確認1');
425                 %disp(range(list_i,1));
```

図 6.75: block_reconstruction.m（９）

```
426                     %disp('確認2');
427 -                    for i=2:range_number(list_i);%範囲から選択（追加もの）
428 -                        range_single=NG;
429                          %disp(range(list_i,i));
430 -                        range_check=OK;
431 -                        if condition(range(list_i,i))==InComplete%条件１
432                              %disp(range(list_i,i));
433 -                            for ip_i=1:taskgraph.task(range(list_i,i)).ip;
434 -                                if condition(taskgraph.task(range(list_i,i)).ip_number(ip_i))~=Complete;
435 -                                    range_check=NG;
436 -                                    break;
437 -                                end
438 -                            end
439 -                            if range_check==OK;%条件２
440                                  %disp(range(list_i,i));
441 -                                for ip_i=1:taskgraph.task(range(list_i,i)).ip;
442 -                                    ip_number1=taskgraph.task(range(list_i,i)).ip_number(ip_i);
443                                      %Pbox(pro_i,2)
444                                      %ip_number1
445 -                                    if P_temp==ip_number1;%臨時Pboxを使用
446                                      %disp(range(list_i,i));
447 -                                        exe_number(range(list_i,i),1)=range(list_i,i);
448 -                                        exe_number(range(list_i,i),2)=pro_i;
449 -                                        exe_number(range(list_i,i),3)=end_time(pro_i);
450 -                                        exe_number(range(list_i,i),4)=taskgraph.task(range(list_i,i)).sum_read_time+
451 -                                        exe_number(range(list_i,i),3);
452 -                                        exe_number(range(list_i,i),5)=taskgraph.task(range(list_i,i)).exetime+exe_number(range(list_i,i),4);
453 -                                        exe_number(range(list_i,i),6)=taskgraph.task(range(list_i,i)).sum_write_time+
454 -                                        exe_number(range(list_i,i),5);
455 -                                        end_time(pro_i)=exe_number(range(list_i,i),6);
456 -                                        condition(range(list_i,i))=-1;%タスクが実行中
457 -                                        check(range(list_i,i))=OK;%table中に入れた
458 -                                        Pbox(pro_i,2)=range(list_i,i)
459 -                                        cannot=0;
460 -                                        finish=0;
461 -                                        total_endtask=total_endtask+1;%完成したタスクが１つ増える
462 -                                        range_single=OK;
463                                          % disp('☆');
464                                          %range(list_i,i)
465
466 -                                    end
467 -                                end
468 -                            end%条件２
469 -                        end%条件1
470 -                        if range_single==OK;
471 -                            break;
472 -                        end
473 -                    end%範囲から選択（追加もの）
474
475 -                if cannot==0;
```

図 6.76: block_reconstruction.m（１０）

```
476 -                 pro_number=pro_number-1;%使えるプロセッサの数が１つ減らす
477 -                 pro_condition(pro_i)=Busy;%そのプロセッサの状態が使えっている状態になった.
478 -                 break;%１つのプロセッサが選んだら循環から跳び出し
479 -                 end
480
481 -                 end%if小循環３_2
482
483
484 -             end%大循環２
485 -         end%大循環１
486
487 -         if cannot==1;%大循環4
488 -         for pro_i=1:PROCESSOR;
489 -             if pro_condition(pro_i)==Free;%プロセッサがちょうど使っていない
490 -                 exe_number(list(list_i),1)=list(list_i);
491 -                 exe_number(list(list_i),2)=pro_i;
492 -                 exe_number(list(list_i),3)=end_time(pro_i);
493 -                 exe_number(list(list_i),4)=taskgraph.task(list(list_i)).sum_read_time+exe_number(list(list_i),3);
494 -                 exe_number(list(list_i),5)=taskgraph.task(list(list_i)).exetime+exe_number(list(list_i),4);
495 -                 exe_number(list(list_i),6)=taskgraph.task(list(list_i)).sum_write_time+exe_number(list(list_i),5);
496
497 -                 end_time(pro_i)=exe_number(list(list_i),6);
498 -                 condition(list(list_i))=-1;%タスクが実行中
499 -                 check(list(list_i))=OK;%table中に入れた
500 -                 Pbox(pro_i,2)=list(list_i)
501 -                 finish=0;
502 -                 total_endtask=total_endtask+1;%完成したタスクが１つ増える
503 -                 pro_number=pro_number-1;%使えるプロセッサの数が１つ減らす
504 -                 pro_condition(pro_i)=Busy;%そのプロセッサの状態が使えっている状態になった.
505 -                 break;%１つのプロセッサが選んだら循環から跳び出し
506 -             end
507 -         end
508 -         end%大循環4
509
510 -         if finish==0;
511 -             break;
512 -         end
513   %************** 「状態3：プロセッサがすべて使えない」 ***********************************
514 -         if pro_number==0;%各プロセッサのend_timeの更新 (98行~119行)
515 -             Min=end_time(1);%任意のプロセッサの終了時間を選ぶ（今のところがプロセッサ１を選び)
516 -             for pro_i=2:PROCESSOR;%プロセッサ１を除いて別のプロセッサを選ぶ, 最小値を確認する.
517 -                 if Min>end_time(pro_i);
518 -                     Min=end_time(pro_i);
519 -                 end
520 -             end
521 -             for pro_i=1:PROCESSOR;
522 -                 if Min==end_time(pro_i);
523 -                     pro_condition(pro_i)=Free;
524 -                     pro_number=pro_number+1;
525 -                     for task_i=2:taskgraph.total_task+1;
```

図 6.77: block_reconstruction.m（１１）

```
526 -                         if condition(task_i)~=Complete;
527 -                             if exe_number(task_i,6)<=Min&&check(task_i)==OK;
528                              %if pro_i==exe_number(task_i,2)&&check(task_i)==OK;
529 -                                condition(task_i)=Complete;
530                                  %task_i
531 -                            end
532 -                        end
533 -                    end
534 -                end
535 -            end
536 -            break;%リストが最初から初期化する
537 -        end
538     %**************「すべて状態完了」**********************************
539 -    end
540 - end
541 - end
542 - end
543   %Pbox
544 - sprintf('  N番号  P番号  endT  読込  実行  書出')
545 - disp(exe_number);
546 - disp(max(exe_number(:,6)));%最大値を表す
547
548   %出力ファイルを作る**********************************************************
549 - fid=fopen('sche.txt','w');
550 - x=1:taskgraph.total_task;%循環=for
551 - y=[exe_number(list(x),:)];%データ配列を作成（各ノードのすべてのデータ）
552 - z=y';%yのデータの（行->列）変換,結果zに代入
553 -     fprintf(fid,'%d %d %d %d %d %d¥n',z);
554
555 - for i=1:taskgraph.total_task;
556 -     data(i,:)=exe_number(list(i),:);
557 - end
558
559   %data
560
561 - end
562
563   %★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
564
565   function ReBlock
566
567       %<<Re-Block method program>>
568   %スケジュール結果の調整プログラム
569   %********①データの読み込み********
570   %data_name=input('dataの名前を入力してください：','s');
571   %fd=fopen(data_name,'r');%ファイルを開く
572   %datas=[];%空の配列を用意
573   %for line_m=1:taskgraph.total_task;%ファイル中のすべてのデータを読み込み
574   %    line=fgets(fd);
575   %temp=str2num(line);
```

図 6.78: block_reconstruction.m（１２）

```
576        %datas=[datas;temp];%配列の追加
577        %end
578        %disp(datas);
579        %disp(datas(41,6));
580 -      program_finish=1;  %★★★★★★★最初はプログラムは実行できると示す.
581 -      OK=1;
582 -      NG=0;
583        %********②スケジューリングを作る********
584 -      list=taskgraph.priority_list;
585 -      PROCESSOR=2;%使用するプロセッサの数（２或いは４）
586 -      scheduling_data=[];
587 -      scheduling_time1=[];
588 -      scheduling_time2=[];
589 -      for pro_i=1:PROCESSOR;
590 -          data_count(pro_i)=0;
591 -      end
592 -      for pro_i=1:PROCESSOR;
593 -          data_temp=0;
594 -              for datas_i=1:taskgraph.total_task;
595 -                  if data(datas_i,2)==pro_i;
596 -                      data_temp=data_temp+1;
597 -                      scheduling_data(pro_i,data_temp)=list(datas_i);
598 -                      scheduling_time1(pro_i,data_temp)=data(datas_i,3);
599 -                      scheduling_time2(pro_i,data_temp)=data(datas_i,6);
600 -                  end
601 -              end
602 -              data_count(pro_i)=data_temp;
603 -      end
604        %scheduling_data
605        %scheduling_time1
606        %scheduling_time2
607        %data_count
608 -      for datas_i=1:taskgraph.total_task;
609 -          for task_i=1:taskgraph.total_task+2;
610 -              if data(datas_i,1)==task_i;
611 -                  scheD(task_i).pro=data(datas_i,2);
612 -                  scheD(task_i).time1=data(datas_i,3);
613 -                  scheD(task_i).time2=data(datas_i,6);
614 -                  break;
615 -              end
616 -          end
617 -      end
618        %%%%%%%%%%④dataの並べる%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
619 -      for pro_i=1:PROCESSOR;
620 -          for sch_i=1:data_count(pro_i);
621 -              min_no=sch_i;
622 -              min_number=scheduling_data(pro_i,sch_i);
623 -              min_time1=scheduling_time1(pro_i,sch_i);
624 -              min_time2=scheduling_time2(pro_i,sch_i);
625 -              for sch_i_temp=sch_i+1:data_count(pro_i);
```

図 6.79: block_reconstruction.m（１３）

```
626 -              if scheduling_time1(pro_i,sch_i_temp)<min_time1;
627 -                  min_no=sch_i_temp;
628 -                  min_number=scheduling_data(pro_i,sch_i_temp);
629 -                  min_time1=scheduling_time1(pro_i,sch_i_temp);
630 -                  min_time2=scheduling_time2(pro_i,sch_i_temp);
631 -              end
632 -          end
633           %%%1代入%
634 -         scheduling_data(pro_i,min_no)=scheduling_data(pro_i,sch_i);
635 -         scheduling_time1(pro_i,min_no)=scheduling_time1(pro_i,sch_i);
636 -         scheduling_time2(pro_i,min_no)=scheduling_time2(pro_i,sch_i);
637           %%%2代出%
638 -         scheduling_data(pro_i,sch_i)=min_number;
639 -         scheduling_time1(pro_i,sch_i)=min_time1;
640 -         scheduling_time2(pro_i,sch_i)=min_time2;
641 -     end
642 - end
643   %scheduling_data
644   %scheduling_time1
645   %scheduling_time2
646   %data_count
647   %*******③再ブロック化法のプログラム<1>隣接及び親子関係のタスクをまとめる********
648   %初期化
649 - R_block1=[];
650 - R_total1=0;
651 - for pro_i=1:PROCESSOR;
652 -     for sche_i=1:data_count(pro_i);
653 -         A=scheduling_data(pro_i,sche_i);
654 -            flag_ip=0;
655 -            for sche_i_temp=sche_i+1:data_count(pro_i);
656 -                remit=0;
657 -                B=scheduling_data(pro_i,sche_i_temp);
658 -                for ip_i=1:taskgraph.task(B).ip;
659 -                    ipnumber_i=taskgraph.task(B).ip_number(ip_i);
660 -                     if ipnumber_i==A;%条件１：２つのタスクが親子関係
661 -                         R_total1=R_total1+1;
662 -                         R_block1(R_total1,1)=A;
663 -                           R_block1(R_total1,2)=B;
664 -                           remit=1;
665 -                           flag_ip=1;
666 -                           break;%親がすでに選ばれたら
667 -                     end
668 -                end
669 -                if remit==0;%１つタスクしかない場合
670 -                   break;
671 -                end
672 -                if flag_ip==1%1回実行した場合
673 -                   break;
674 -                end
675 -            end
```

図 6.80: block_reconstruction.m（１４）

```
676 -          end
677 -      end
678        %R_block1%ブロック可能な配列
679        %R_total1%ブロック可能な数
680        %scheD(42)
681        %*******③再ブロック化法のプログラム<2>. 上の<1>に, 2つの条件を満たすタスクを求める********
682 -          R_block2=[];
683 -          R_total2=0;
684 -      for R_i=1:R_total1;
685 -          condition1=1;
686 -          condition2=1;
687 -              for ip_i=1:taskgraph.task(R_block1(R_i,2)).ip;%ブロック可能なタスクは親と比べる
688 -                  ipnumber_i=taskgraph.task(R_block1(R_i,2)).ip_number(ip_i);
689 -                  if scheD(ipnumber_i).pro~=scheD(R_block1(R_i,2)).pro & ipnumber_i~=1;
690 -                      if scheD(ipnumber_i).time2>scheD(R_block1(R_i,1)).time1;%条件2.1の否の場合
691 -                          condition1=0;
692 -                          break;
693 -                      end
694 -                  end
695 -              end
696 -              for is_i=1:taskgraph.task(R_block1(R_i,1)).is;%ブロック可能なタスクは子と比べる
697 -                  isnumber_i=taskgraph.task(R_block1(R_i,1)).is_number(is_i);
698 -                  if isnumber_i~=taskgraph.total_task+2;
699 -                  if scheD(isnumber_i).pro~=scheD(R_block1(R_i,1)).pro; %&& isnumber_i~=taskgraph.total_task+2;
700 -                      if scheD(isnumber_i).time1<scheD(R_block1(R_i,1)).time2-2;%条件2.2の否の場合
701 -                          condition2=0;
702 -                          break;
703 -                      end
704 -                  end
705 -                  end
706 -              end
707 -          if condition1==1 && condition2==1;
708 -              R_total2=R_total2+1;
709 -              R_block2(R_total2,1)=R_block1(R_i,1);
710 -              R_block2(R_total2,2)=R_block1(R_i,2);
711 -          end
712 -      end
713        %R_block2
714        %R_total2
715        %*******④同じタスクを持っているブロックを削除********
716        %初期化
717 -      for task_i=2:taskgraph.total_task+1;%初期化 (sとtがチェックしない)
718 -          R_complete(task_i)=0;
719 -      end
720 -      block=[];
721 -      total_block=0;
722 -          for R_i=1:R_total2;
723 -              if R_complete(R_block2(R_i,1))==0;
724 -                  total_block=total_block+1;
725 -                  block(total_block,1)=R_block2(R_i,1);
```

図 6.81: block_reconstruction.m（１５）

```
726 -                block(total_block,2)=R_block2(R_i,2);
727 -                R_complete(R_block2(R_i,1))=1;
728 -                R_complete(R_block2(R_i,2))=1;
729 -            end
730 -        end
731 -        fprintf('ブロックに含まれるノード:¥n');
732 -        block%最終結果
733 -        total_block
734
735 -        if total_block==0;%★★★★★★★groupの数のチェック
736 -            program_finish=0;
737 -            disp('★★★★★このタスクグラフにブロック化法できない★★★');
738 -            disp('★★★ブロックの数は"0"ので，プログラムそのまま終了★★★');
739 -        end
740
741     %*******⑤タスクグラフの変更（基本ブロック化方と追加ブロック化方とちょっと違い）********
742
743     %%%%%%%%%%%%%%%%ブロックが新たなノードに変換し，その親の情報<2>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
744 -    if program_finish==1;%★★★★★★★もし探したgroupの数は0にならば，プログラム終了(groupとは組み合わせしていないブロックです)
745 -    for block_i=1:total_block;
746 -    for task_i=1:taskgraph.total_task+2;%すべてのノードが初期化
747 -        condition(task_i)=NG;
748 -    end
749 -    condition(block(block_i,1))=OK;
750 -    for ip_i=1:taskgraph.task(block(block_i,1)).ip;
751 -        condition(taskgraph.task(block(block_i,1)).ip_number(ip_i))=OK;
752 -    end
753
754 -        taskgraph.task(block(block_i,1)).exetime=taskgraph.task(block(block_i,1)).exetime+taskgraph.task(block(block_i,2)).exetime;
755 -        condition(block(block_i,2))=OK;
756
757 -        for ip_i_temp=1:taskgraph.task(block(block_i,2)).ip;
758 -            if condition(taskgraph.task(block(block_i,2)).ip_number(ip_i_temp))~=OK;
759 -                ip_i=ip_i+1;
760 -                taskgraph.task(block(block_i,1)).ip_number(ip_i)=taskgraph.task(block(block_i,2)).ip_number(ip_i_temp);
761 -                taskgraph.task(block(block_i,1)).ip_read(ip_i)=taskgraph.task(block(block_i,2)).ip_read(ip_i_temp);
762 -                taskgraph.task(block(block_i,1)).ip=taskgraph.task(block(block_i,1)).ip+1;
763 -                condition(taskgraph.task(block(block_i,2)).ip_number(ip_i_temp))=OK;
764 -            end
765 -        end
766
767 -    for task_i=1:taskgraph.total_task+2;%すべてのノードが初期化
768 -        condition(task_i)=NG;
769 -    end
770 -    end
771     %%%%%%%%%%%%%%%%%%%%%すべてのノードの親の情報の更新<3>%%%%%%%%%%%%%%%%%%%%%
772 -    for task_i=1:taskgraph.total_task+2;
773 -        for ip_i=1:taskgraph.task(task_i).ip;
774 -            for block_i=1:total_block;
775 -                if taskgraph.task(task_i).ip_number(ip_i)==block(block_i,2);
```

図 6.82: block_reconstruction.m（１６）

```
776 -                 taskgraph.task(task_i).ip_number(ip_i)=block(block_i,1);
777 -             end
778 -         end
779 -     end
780 - end
781   %%%%%%%%%%%%%%%%%%%%%%%%%要らないノードの情報を捨てる<4>%%%%%%%%%%%%%%%%%%%%%%%
782 - for block_i=1:total_block;
783 -         taskgraph.task(block(block_i,2)).exetime=0;
784 -         taskgraph.task(block(block_i,2)).ip=0;
785 -         taskgraph.task(block(block_i,2)).ip_number=0;
786 -         taskgraph.task(block(block_i,2)).ip_read=0;
787 -         taskgraph.task(block(block_i,2)).is=0;
788 -         taskgraph.task(block(block_i,2)).sum_write_time=0;
789 -         taskgraph.task(block(block_i,2)).sum_read_time=0;
790 -         taskgraph.task(block(block_i,2)).is_number=0;
791 -         taskgraph.task(block(block_i,2)).pre=0;
792 -         taskgraph.task(block(block_i,2)).pre_number=0;
793 -         taskgraph.task(block(block_i,2)).suc=0;
794 -         taskgraph.task(block(block_i,2)).suc_number=0;
795 -         taskgraph.task(block(block_i,2)).pre_number_temp=0;
796 -         taskgraph.task(block(block_i,2)).suc_number_temp=0;
797 - end
798   %%%%%%%%%%%%%%%%%%%%%%%%%%親のデータの並び（昇）<7>%%%%%%%%%%%%%%%%%%%%%%%(通信時間と親がsort)
799 - for task_i=1:taskgraph.total_task+2;
800 -     for ip_i=1:taskgraph.task(task_i).ip;
801 -         min_no=ip_i;
802 -         min_number=taskgraph.task(task_i).ip_number(ip_i);
803 -         min_read=taskgraph.task(task_i).ip_read(ip_i);
804 -         for ip_i_temp=ip_i+1:taskgraph.task(task_i).ip;
805 -             if taskgraph.task(task_i).ip_number(ip_i_temp)<min_number;
806 -                 min_no=ip_i_temp;
807 -                 min_number=taskgraph.task(task_i).ip_number(ip_i_temp);
808 -                 min_read=taskgraph.task(task_i).ip_read(ip_i_temp);
809 -             end
810 -         end
811 -         taskgraph.task(task_i).ip_number(min_no)=taskgraph.task(task_i).ip_number(ip_i);
812 -         taskgraph.task(task_i).ip_read(min_no)=taskgraph.task(task_i).ip_read(ip_i);
813 -         taskgraph.task(task_i).ip_number(ip_i)=min_number;
814 -         taskgraph.task(task_i).ip_read(ip_i)=min_read;
815 -     end
816 - end
817   %%%%%%%%%%%%%%%%%%%%%%%%%同じ結果を削除(親)<8>%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
818 - for task_i=2:taskgraph.total_task+2;
819 -     if taskgraph.task(task_i).ip~=0;%情報なしのノード情報が実行しない
820 -     p=taskgraph.task(task_i).ip_number(1);
821 -     task_sample(task_i).ip_number_temp(1)=taskgraph.task(task_i).ip_number(1);
822 -     task_sample(task_i).ip_read_temp(1)=taskgraph.task(task_i).ip_read(1);
823 -     j=0;%削除した数
824 -     for i=2:taskgraph.task(task_i).ip;
825 -         if p==taskgraph.task(task_i).ip_number(i);
```

図 6.83: block_reconstruction.m（１７）

```
826 -                 j=j+1;
827 -             else
828 -                 task_sample(task_i).ip_number_temp(i-j)=taskgraph.task(task_i).ip_number(i);
829 -                 task_sample(task_i).ip_read_temp(i-j)=taskgraph.task(task_i).ip_read(i);
830 -                 p=taskgraph.task(task_i).ip_number(i);
831 -             end
832 -         end
833 -         taskgraph.task(task_i).ip=taskgraph.task(task_i).ip-j;
834 -         end%if文のendだ！，情報なしノードがこのプログラムを参加さいない
835 -     end
836 -     for task_i=2:taskgraph.total_task+2;
837 -         taskgraph.task(task_i).ip_number=task_sample(task_i).ip_number_temp;
838 -         taskgraph.task(task_i).ip_read=task_sample(task_i).ip_read_temp;
839 -     end
840       %%%%%%%%%%%%%%%%%%%%%%ブロック後のtaskgraphデータを整理する<9>%%%%%%%%%%%%%%%%%
841 -     graph=[];
842       %graph(1,1)=taskgraph.total_task;
843 -     for graph_i=1:taskgraph.total_task+2;%sとtとgraph(1,1)
844 -         graph(graph_i,1)=taskgraph.task(graph_i).task_number;
845 -         graph(graph_i,2)=taskgraph.task(graph_i).exetime;
846 -         graph(graph_i,3)=taskgraph.task(graph_i).ip;
847 -         graph_j=4;
848 -         for ip_i=1:taskgraph.task(graph_i).ip;
849 -             graph(graph_i,graph_j)=taskgraph.task(graph_i).ip_number(ip_i);
850 -             graph_j=graph_j+1;
851 -             graph(graph_i,graph_j)=taskgraph.task(graph_i).ip_read(ip_i);
852 -             graph_j=graph_j+1;
853 -         end
854 -     end
855       %graph
856       %break;%ブロックらが１つを実行だけ.
857       %    end%redundant(完)
858        %   if block_i==total_block;%(プログラム終了条件)
859         %      exit_program=1;
860          % end
861       %end%block循環(完)
862       %%%%%%%%%%%%%%%%%%%%%%graphの出力1%%%%%%%%%%%%%%%%%
863 -     graph_out=[];
864 -     graph_out(1,1)=taskgraph.total_task;
865 -     for graph_i=2:taskgraph.total_task+3;%sとtとgraph(1,1)
866 -         graph_out(graph_i,1)=taskgraph.task(graph_i-1).task_number;
867 -         graph_out(graph_i,2)=taskgraph.task(graph_i-1).exetime;
868 -         graph_out(graph_i,3)=taskgraph.task(graph_i-1).ip;
869 -         graph_j=4;
870 -         for ip_i=1:taskgraph.task(graph_i-1).ip;
871 -             graph_out(graph_i,graph_j)=taskgraph.task(graph_i-1).ip_number(ip_i);
872 -             graph_j=graph_j+1;
873 -             graph_out(graph_i,graph_j)=taskgraph.task(graph_i-1).ip_read(ip_i);
874 -             graph_j=graph_j+1;
875 -         end
```

図 6.84: block_reconstruction.m（１８）

```
876 -   end
877 -   graph
878     %%%%%%%%%%%%%%%%%%%%%%graphの出力2%%%%%%%%%%%%%%%%%
879 -   for task_i=1:taskgraph.total_task+2;
880 -       checkzero(task_i)=0;
881 -   end
882 -   checkzero(1)=1;
883 -   newtotal=0;
884
885 -   for task_i=2:taskgraph.total_task+2;
886 -       if taskgraph.task(task_i).ip~=0;
887 -           newtotal=newtotal+1;
888 -          checkzero(task_i)=1;
889 -       end
890 -   end
891 -   newtotal=newtotal-1;%tノードを除く
892     %初期化終了
893 -   graph_out1=[];
894 -   graph_out1(1,1)=newtotal;
895 -   graph_ii=2;
896 -   for graph_i=2:taskgraph.total_task+3;%sとtとgraph(1,1)
897 -       if checkzero(graph_i-1)==1;
898 -       graph_out1(graph_ii,1)=taskgraph.task(graph_i-1).task_number;
899 -       graph_out1(graph_ii,2)=taskgraph.task(graph_i-1).exetime;
900 -       graph_out1(graph_ii,3)=taskgraph.task(graph_i-1).ip;
901 -       graph_j=4;
902 -       for ip_i=1:taskgraph.task(graph_i-1).ip;
903 -           graph_out1(graph_ii,graph_j)=taskgraph.task(graph_i-1).ip_number(ip_i);
904 -           graph_j=graph_j+1;
905 -           graph_out1(graph_ii,graph_j)=taskgraph.task(graph_i-1).ip_read(ip_i);
906 -           graph_j=graph_j+1;
907 -       end
908 -       graph_ii=graph_ii+1;
909 -       end
910 -   end
911
912
913     %出力ファイル1を作る*********************************************************
914 -   fid=fopen('Result_Graph.txt','w');
915 -   for x=1:taskgraph.total_task+3;%循環=for
916 -       y=[graph_out(x,:)];
917 -       z=y';%yのデータの（行->列）変換,結果zに代入
918 -           fprintf(fid,'%d ',z);%一行の内容が表示できる
919 -           fprintf(fid,' ¥n',z);%最後に新たな行に変更
920 -   end
921     %出力ファイル2を作る(ノードの情報がない場合表示しない)*********************************************************
922 -   fid1=fopen('New_Graph.txt','w');
923 -   for x=1:newtotal+3;%循環=for
924 -       y=[graph_out1(x,:)];
925 -       z=y';%yのデータの（行->列）変換,結果zに代入
```

図 6.85: block_reconstruction.m（１９）

```
926 -            fprintf(fid1,'%d ',z);%一行の内容が表示できる
927 -            fprintf(fid1,' ¥n',z);%最後に新たな行に変更
928 -     end
929
930
931 -     end%(program_finish)
932 -     end%(ReBlock)
933
934 -     end
```

図 6.86: block_reconstruction.m（２０）

（4）次は優先リスト順位に基づいてタスクグラフのスケジューリングのプログラムを示す．本プログラムの実行ファイルは Scheduling_1.m という名前のスクリプト M-ファイルである．このプログラムを利用して，タスクグラフのスケジュール結果を求められる．

```
1      %*********************<<スケジューリングのプログラム>>*********************
2      %(7)スケジューリングのプログラムを作成
3      %disp(taskgraph);
4      %****************************<<実行準備>>**********************************
5  -   list=taskgraph.priority_list;
6  -   PROCESSOR=2;%使用するプロセッサの数（２或いは４）
7  -   Complete=1; %（ノードの状態)
8  -   InComplete=0; %（ノードの状態)
9  -   OK=1; %（チェックの状態)
10 -   NG=0; %（チェックの状態)
11 -   Free=1; %（プロセッサの状態)
12 -   Busy=0; %（プロセッサの状態)
13     %**************************************************************************
14     %初期化（プロセッサの最初状態と各タスクの最初状態)
15 -   for pro_i=1:PROCESSOR;
16 -       pro_condition(pro_i)=Free;%プロセッサを使っていない
17 -       end_time(pro_i)=0;%各プロセッサの最短時刻
18 -   end
19 -   for task_i=1:taskgraph.total_task+2;
20         %flag2(task_i)=OK;
21 -       condition(task_i)=InComplete;
22 -   end
23 -   condition(1)=Complete;%ｓノードは完成
24 -   total_endtask=0;%チェック完成したノードの数
25 -   flaglost=NG;%リストの指針は最後のデータを指していない場合
26 -   pro_number=PROCESSOR;%残したプロセッサの数を表示
27 -   exe_number=zeros(taskgraph.total_task+2,6);%exe_number配列を用意
28     %実行開始********************************************************************
29 -   while total_endtask~=taskgraph.total_task;%総の循環（完成したノード=総ノード数)
30 -   for list_i=1:taskgraph.total_task;%listの循環
31 -       flagend=NG;
32 -       if list(list_i)==list(taskgraph.total_task);%指針がリストの最後データを指したら
33 -           flaglost=OK;
34 -       end
35 -   check1=OK;%ノードの親がチェック完成したどうか
36 -   if condition(list(list_i))==InComplete;%探索開始
37 -       for ip_i=1:taskgraph.task(list(list_i)).ip;%各プロセッサのend_timeの更新（37行~77行)
38 -           ip_number_temp=taskgraph.task(list(list_i)).ip_number(ip_i);
39     %************* 「状態１：プロセッサがBusy||Free，実行できるノード完全がない」************
40 -           if condition(ip_number_temp)~=Complete;%親のチェック
```

図 6.87: Scheduling_1.m（１）

```
41 -             check1=NG;
42 -             if flaglost==OK;%リストの中で選ぶ相手がない時に（リストの最後に指す)
43 -                 Min_temp=9999;
44 -                 for pro_i=1:PROCESSOR;%一番最小のプロセッサ実行値（Min_temp）を確認
45 -                     if pro_condition(pro_i)~=Free;%('使っている'プロセッサだけ選ぶ)
46 -                     if Min_temp>end_time(pro_i);
47 -                         Min_temp=end_time(pro_i);
48 -                     end
49 -                     end
50 -                 end
51 -                 for pro_i=1:PROCESSOR;
52 -                     if Min_temp==end_time(pro_i);%最小値を持っているプロセッサの番号を確認
53 -                         pro_condition(pro_i)=Free;%そのプロセッサが実行完了と設定する.
54 -                         pro_number=pro_number+1;
55 -                         for task_i=1:taskgraph.total_task+2;%実行完了プロセッサがどのタスクを実行する？って確認する
56 -                             if condition(task_i)~=Complete;
57 -                                 if pro_i==exe_number(task_i,2)&&check(task_i)==OK;%taskのデータはexe_numberの配列に入り（OKになる)
58 -                                     condition(task_i)=Complete;%違いプロセッサのデータを比較して，endtime少ない方がチェック完成
59 -                                     flaglost=NG;%実行ノードがありになった
60 -                                     flagend=OK;%優先リストのためにデータ１回実行した後で，優先度高いデータを選ぶ
61 -                                 end
62 -                             end
63 -                         end
64 -                     end
65 -                 end
66 -                 for pro_i=1:PROCESSOR;%各プロセッサの最後の時刻が一緒にする（短いプロセッサが長いプロセッサに一致する)
67 -                     if end_time(pro_i)<Min_temp;
68 -                         end_time(pro_i)=Min_temp;
69 -                     end
70 -                 end
71 -             end%p42終了
72 -             break;%親の循環に跳び出し
73 -         end
74 -     end
75 -     if flagend==OK;%リストがもう一度最初から（初期化)
76 -         break;
77 -     end
78    %**************「状態2：普通の状態」**********************************
79 -     if check1==OK;%親がすべてCompleteならば,データを代入
80 -         for pro_i=1:PROCESSOR;
81 -             if pro_condition(pro_i)==Free;%プロセッサがちょうど使っていない
82 -                 exe_number(list(list_i),1)=list(list_i);
83 -                 exe_number(list(list_i),2)=pro_i;
84 -                 exe_number(list(list_i),3)=end_time(pro_i);
85 -                 exe_number(list(list_i),4)=taskgraph.task(list(list_i)).sum_read_time+exe_number(list(list_i),3);
86 -                 exe_number(list(list_i),5)=taskgraph.task(list(list_i)).exetime+exe_number(list(list_i),4);
87 -                 exe_number(list(list_i),6)=taskgraph.task(list(list_i)).sum_write_time+exe_number(list(list_i),5);
88
89 -                 end_time(pro_i)=exe_number(list(list_i),6);
90 -                 condition(list(list_i))=-1;%タスクが実行中
```

図 6.88: Scheduling_1.m（２）

```
 91 -                 check(list(list_i))=OK;%table中に入れた
 92 -                 total_endtask=total_endtask+1;%完成したタスクが１つ増える
 93 -                 pro_number=pro_number-1;%使えるプロセッサの数が１つ減らす
 94 -                 pro_condition(pro_i)=Busy;%そのプロセッサの状態が使えっている状態になった.
 95 -                 break;%１つのプロセッサが選んだら循環から跳び出し
 96 -             end
 97 -         end
 98     %**************「状態3：プロセッサがすべて使えない」***********************************
 99 -         if pro_number==0;%各プロセッサのend_timeの更新 (98行~119行)
100 -             Min=end_time(1);%任意のプロセッサの終了時間を選ぶ (今のところがプロセッサ１を選び)
101 -             for pro_i=2:PROCESSOR;%プロセッサ１を除いて別のプロセッサを選ぶ，最小値を確認する.
102 -                 if Min>end_time(pro_i);
103 -                     Min=end_time(pro_i);
104 -                 end
105 -             end
106 -             for pro_i=1:PROCESSOR;
107 -                 if Min==end_time(pro_i);
108 -                     pro_condition(pro_i)=Free;
109 -                     pro_number=pro_number+1;
110 -                     for task_i=1:taskgraph.total_task+2;
111 -                         if condition(task_i)~=Complete;
112 -                             if pro_i==exe_number(task_i,2)&&check(task_i)==OK;
113 -                                 condition(task_i)=Complete;
114 -                             end
115 -                         end
116 -                     end
117 -                 end
118 -             end
119 -             break;%リストが最初から初期化する
120 -         end
121      %**************「すべて状態完了」***********************************
122 -     end
123 - end
124 - end
125 - end
126 - sprintf('  N番号  P番号  endT  読込  実行  書出')
127 - disp(exe_number);
128 - disp(max(exe_number(:,6)));%最大値を表す
129   %出力ファイルを作る**********************************************************
130 - fid=fopen('sche.txt','w');
131 - x=1:taskgraph.total_task;%循環=for
132 - y=[exe_number(list(x),:)];%データ配列を作成 (各ノードのすべてのデータ)
133 - z=y';%yのデータの (行->列) 変換,結果zに代入
134 -     fprintf(fid,'%d %d %d %d %d %d¥n',z);
```

図 6.89: Scheduling_1.m（3）

（5）最後はクリティカルパス (CP) のプログラムを示す．本プログラムの実行ファイルは CP.m という名前のスクリプト M-ファイルである．ブロック化法の有効性を評価することには無限のプロセッサが必要である．ブロック化法のシミュレーションするためにこのプログラムを利用する．

```
%**********************<<最長パスのプログラム>>**********************

%初期化
for condition_i=1:taskgraph.total_task+2;
    condition(condition_i)=0;
    taskgraph.task(condition_i).srlen=0;
end
condition(taskgraph.total_task+2)=1;
end_task=1;

while end_task<taskgraph.total_task+2;
    for task_i=1:taskgraph.total_task+2;
        check=1;
        if condition(task_i)==0;
            for is_i=1:taskgraph.task(task_i).is;
                is_number_temp=taskgraph.task(task_i).is_number(is_i);
                if condition(is_number_temp)~=1;
                    check=0;
                    break;
                end
            end
            if check==1;
                max_srlen=0;
                for is_i=1:taskgraph.task(task_i).is;
                    is_number_temp=taskgraph.task(task_i).is_number(is_i);
                    max_srlen_temp=taskgraph.task(is_number_temp).srlen;
                    if max_srlen<=max_srlen_temp;
                        max_srlen=max_srlen_temp;
                    end
                end
                taskgraph.task(task_i).srlen=max_srlen+taskgraph.task(task_i).sum_read_time+taskgraph.task(task_i).sum_write_time;
                end_task=end_task+1;
                condition(task_i)=1;
            end
        end
    end
end

taskgraph.cp_length=taskgraph.task(1).srlen;
fprintf('CP長： %d  ¥n', taskgraph.cp_length);
for task_i=1:taskgraph.total_task+2;
    fprintf('%d の srlen： %d  ¥n', task_i, taskgraph.task(task_i).srlen);
end
```

図 6.90: CP.m