卒業研究報告書

題目

BSP モデル上での Selection プログラム

指 導 教 員

石水 隆 助手

報告者

02-1-47-084

中内 義典

近畿大学理工学部情報学科

平成 19 年 2 月 22 提出

概要

本研究では、BSP(Bulk-Synchronous Parallel)モデル上で選択問題(Selection)を解くアルゴリズムの正当性および、その計算量を実験的に評価するため、シミュレートプログラムを作成する。

BSP モデルは Valiant により提案された非同期式分散メモリ型並列計算モデルであり、並列計算において重要とされる通信コストを同期時間 L 通信命令実行時間 g といったパラメタにより表すことを可能としたモデルである

BSP モデルはクラスタ処理（Cluster Computing）およびグリッド処理（Grid Computing）による並列化に対応したモデルとして注目されており、そのため多くの関数に対して BSP モデル上で効率よく実行できる並列アルゴリズムが求められている。

目次

第１章　序論

1.1 並列処理

通常１人で行なう処理を、何人かで処理する量を分割して、同時に行うのが並列処理(Parallel Processing)である。例えば、１００個のランダムな数字を評価して一番小さい数字を１つ選ぶのに、評価する人間が２人いれば、１人ですべて評価するときの約半分の時間で評価できるだろうし、評価する人間が３人に増えれば約３分の１の時間で評価を終えることは想像できる。しかし、評価する人間が１００人いるとすると１００分の１の時間で評価を終えることができるだろうか。少なくとも、どの範囲を誰が評価するか当てはめたり、評価結果同士を比較したりし一番小さい数字を探すのにも時間がかかるであろうし、１００分の１で終わることは到底考えられない。計算機で並列計算処理を行う場合でも、同じような問題が起こる。

このように、処理を分割したがために新たに必要となる（本来は必要不要な）処理をオーバーヘッドと言う。２人や３人で処理をするときにはほとんど問題にはならないが、５０人、６０人・・と増やしていくとオーバーヘッドは目立ってくる。要するに、全体の作業量と作業者の数のバランスが重要である。計算機の世界では、作業量とは処理すべきデータ量（問題規模）で、作業者とはプロセッサと当てはめられる。

地球環境のシミュレーション, 天気予報で用いられる数値予報,宇宙物理のシミュレーション, 流体の計算, コンピュータグラフィックス、画像処理, 大規模データベース等, 従来の逐次型計算機では扱うのが困難な大規模かつ複雑な問題を解く必要性から, 並列計算機への移行が行われ、またスーパーコンピュータや大規模サーバなどはすべて並列型になっている。身近なパソコンにも同じことが言える。つまり並列計算なしにはこれからのコンピュータ機器の発展は望めない。

1.2　並列計算機

並列計算機(Parallel Computer)は複数のプロセッサを持ち並列処理を行うことができる計算機である。並列計算機は、全てのプロセッサが共通したメモリに対して読み書きを行い、プロセッサ間の通信はメモリを通して行う共有メモリ型並列計算機（Shared Memory Parallel Computer）と、それぞれのプロセッサが局所メモリを持ち、プロセッサ間の通信はネットワークを通じて行う分散メモリ型並列計算機（Distributed Memory Parallel Computer）に大別される。共有メモリ型計算機はプロセッサ間の通信を高速に行うことができ、プロセッサ間での同期も取り易いため、通信および同期にかかるコストを気にせずに高速化を得ることができるが、プロセッサ数を増やすことは困難である。逆に分散メモリ型計算機は比較的多数のプロセッサを持つことができるが、プロセッサ間の通信には時間がかかる。

このため、プロセッサ数が少ない並列計算機は共有メモリ型が、プロセッサ数が多い並列計算機は分散メモリ型が主流になっている。

1.3　並列アルゴリズム

アルゴリズム(Algorithm) とは、ある目的を実現するために必要な作業の論理や手順を、明確に述べたものである。この手順とは、でたらめにやるのではなく、その指示どおりに正確に従い記述することである。例えば、二つの整数が与えられたときに、筆算でこれらから商と余りを求める方法などは、アルゴリズムの例である。また、料理のレシピ等も、ある決まった料理を作るために、指定された種類と分量の材料を使って、決まった手順で作業を行うという点では、広い意味のアルゴリズムに含まれる。

ある問題を解くアルゴリズムの評価にはさまざまな方法がある。1つには、プログラムの短さや簡単さといった基準が考えられるが、この方法では、肝心の計算効率が評価されない。おそらく、計算時間を評価する一番分かりやすい方法は、実際にプログラムを書いてそれを実行させてみることである。しかし、この方法では、様々なアルゴリズムを比較するには、同じ計算機で実行しなければならないことに

なる。さらに困ったことには、ある計算機ではもっとも早く動作したプログラムが、別の計算機では他のプログラムより遅いということも起こり得る。

そこで、実際の計算機の複雑な部分を考えずにすむように、アルゴリズムの効率の解析では、計算機のモデルを考えてその上で評価する方法が用いられる。

計算量、つまり計算効率の評価の基準として通常用いられるのは、計算時間と記憶領域である。並列計算モデルの場合はそれに加え、プロセッサ数が評価の対象となることが多い。莫大なメモリやプロセッサ数を必要とするアルゴリズムでは事実上計算機で解けなくなることから、計算時間だけでなく、これらも重要な評価の尺度となっている。このほか、回路や VLSI としての実現に基づくモデルでは、これらに対応する、回路の段数やサイズ、面積などが用いられる。

並列アルゴリズム(Parallel Algorithm)は、並列計算モデル上で実行させるためのアルゴリズムである。並列アルゴリズムは、与えられた問題をどのようによりサイズの小さい部分問題に分割し、それをどのように各プロセッサに割り当てるかを記述せねばならない。そのため、一般的には逐次アルゴリズムをそのまま並列計算モデル上で動かすことはできず、並列計算モデル用に新たにアルゴリズムを作成する必要がある。

1.4 並列計算モデル

並列アルゴリズムの設計および解析は並列計算機を抽象化して並列計算モデル(Parallel Computing Model)上で行なわれる。代表的な並列計算モデルは PRAM(Parallel Random Access Machine)、Mesh、Hyper Cube BSP(Bulk-Synchronous Parallel)などがある。

1.5 PRAM

共有メモリとそれに接続された複数のプロセッサから成る並列計算機を共有メモリ型並列計算機と呼ぶ。PRAM(Parallel Random Access Machine)[2]は共有メモリ型並列計算機を抽象化したモデルである

初期の並列計算機の多くが低ビットの並列処理専用に開発されたプロセッサを使って、短い同期を取りながら処理を行うものが多かったため、従来の並列アルゴリズムの設計・解析は一般的に PRAM (Parallel Random Access Machine) 上で行われる。PRAM は共有メモリ型モデルであり、通信や同期に掛かるコストを無視でき、また、1 命令ごとに全てのプロセッサで同期を取る最粒度同期式モデルであるため、並列アルゴリズムを設計・解析を行い易い。

PRAM 上の並列アルゴリズムの実行中、各プロセッサは入力データの読み出し、中間結果の読み出し及び書き込み、最終結果の書き出しのため共有メモリにアクセスする。PRAM は、各プロセッサは共有メモリ上の任意の位置にあるメモリセルに対して、1 単位時間でデータの読み書きができ、また全ての演算は 1 単位時間でできると仮定されている。また 1 単位時間に全てのプロセッサで同期が取られる完全同期型モデルである。

共有メモリ型であるため、PRAM ではデータの局所性は考慮されない。また、プロセッサ間の通信は共有メモリを通して行なわれること、完全同期型であることから、プロセッサ間通信や同期にかかる遅延は無視されている

しかし、プロセッサ能力の向上に伴い、プロセッサ間の通信時間が並列アルゴリズムの実行時間の大きな割合を占めるようになってきた。また、プロセッサが他のプロセッサと同期せずに処理を行う非同期式処理も主流となり、これらの特徴を持つ並列計算機に対しては PRAM ではアルゴリズムの性能を正確に評価することが困難になってきている。

1.6 BSP モデル

前節で述べた理由から BSP (Bulk-Synchronous Parallel) モデル[3]がプロセッサ内の通信時間が実行時間全体に占める割合が大きい並列計算機に対応させるために提案された。BSP モデルは分散メモリ型の非同

期式並列計算モデルであり、通信や同期のコストはパラメタにより抽象化されている、このため BSP モデルは多くの実在する並列計算機に対応する汎用性高いモデルである。

1.7 本研究の目的

本研究では BSP モデル上で選択問題(Selection)を解く並列アルゴリズムに対して、その正当性および計算量の実験的な評価を行なうためシミュレートプログラムを作成する。

シミュレートプログラムにより、プロセッサ数、データ数の値をかえていった場合、計算時間がどのように変化するか、また Selection によってどれだけ計算時間が速くなるのかを検討する

第２章　準　備

2.1　BSP モデル

BSP(Bulk-Synchronous Parallel)モデル[3]は非同期式並列計算モデルであり、以下の構成要素から成る。

・局所メモリを持つ複数のプロセッサ
・プロセッサ間の１対１メッセージ通信を行なう完全結合網
・プロセッサ間の同期を実現するための同期機構

BSP モデルではバリア同期以外の同期は無いという非常に緩い同期を仮定している
図 2.1 に BSP モデルの概念図を示す

また、BSP モデルは以下のパラメタを持つ

p:プロセッサ数

g：通信路帯域幅：１個のメッセージを送信あるいは受信するためにかかる時間。

L：同期時間：プロセッサ全体の１回の同期にかかる時間、これは同時に通信遅延時間を表すパラメタでもある

BSP モデルでは、１つの内部計算命令の実行に１単位時間掛かるとき、１つの通信命令の実行に g 時間、１回の同期に L 時間掛かると仮定されている。

BSP モデル上での並列アルゴリズムは、各プロセッサが実行するプログラムにより表される。各プロセッサが実行するプログラムは**スーパーステップ**の列からなる。各スーパーステップは内部計算命令の列からなる内部計算フェーズと、送信命令、受信命令の列からなる通信フェーズで構成されており、各プロセッサはスーパーステップの命令を非同期に実行する。

また、スーパーステップの命令を終了後、プロセッサ間でバリア同期を取り、次のスーパーステップの実行に移る。メッセージの受信については、各スーパーステップ中の通信フェーズで送信されたメッセージは同一のスーパーステップの通信で受信されるが、そのメッセージはその次のスーパーステップ以降でしか利用できないものと仮定する。あるスーパーステップで各プロセッサ P_i($1 \leqq i \leqq P$)がそれぞれｗ個の内部計算命令とｈ個の送信命令あるいは受信命令を実行するとき、そのスーパーステップ全体の実行時間Ｏ（ｗ＋ｇｈ＋Ｌ）時間かかる

スーパーステップの特性を持つことにより特性を保たち、データ通信の依存関係の解析ができる。

2.2　選択問題

選択問題（Selection）とはサイズｎ配列 A および整数 ｄ(0≦d<n)が与えられたとき、データ中の d 番目に小さい要素を探し出す問題である。BSP モデル上では、選択問題を

$$O(gn/p + L \log p) \quad \text{.....(1)}$$

時間で解く事ができる

2.3　BSP モデル上での Selection アルゴリズム

選択問題を解くアルゴリズムは以下のステップから成る。初期状態において、各プロセッサ P_i (0≦i<p) は A のサイズ n/p の部分配列 A_i を保持する。

1.　以下の 1.1～1.6 を log p 回繰り返す
1.1.　P_i (0≦i<p) は A_i の中央値 m_i を求め、P_0 に送信する
1.2.　P_0 は m_i の中央値 m を求め、全てのプロセッサに送信する。
1.3.　P_i は A_i を m よりも小さい部分配列 A_i^L と A_i^R に分割する。

1.4. P_iはP_0にA_i^Lの要素数を送信する。
1.5. P_0は求めるべき要素がA_i^LとA_i^Rのどちらにあるか判定し、他のプロセッサに伝える。
1.6. P_iは求めるべき要素が含まれて居ない部分配列を破棄し、残りの部分配列を新たなA_iとする。
2. P_iは保持するデータをP_0に送信する。
3. P_0は逐次に選択問題を解き解を出力する。

本研究では、上記のアルゴリズムをシミュレートするプログラムを JAVA を用いて作成し、各パラメタが変化したときの実行時間の測定を行い、理論値(式(1))との比較を行う。

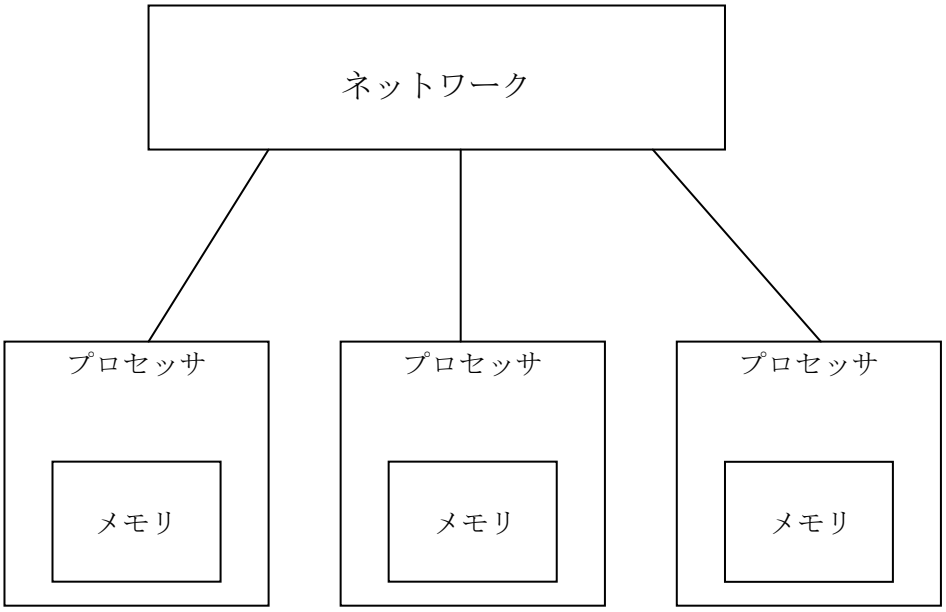


図 2.1：BSP モデルの概念図

第３章　方　法

3.1　BSP 上での Selection アルゴリズム

本研究では JAVA 言語を用い、BSP 上で Selection アルゴリズムの実行をシミュレートするプログラムを作成した。本研究で作成したプログラムは、データ数 n、プロセッサ数 p、通信帯域幅 g、同期時間 L が与えられたとき、データ数 n の中で d 番目に小さい要素を探し出すことができる。また、選択問題アルゴリズムと BSP モデル上で実行したとき、実行時間を測定できる

3.2　実行時間の測定

通信遅延、通信コスト、データ数、プロセッサ数を変化させ、内部計算時間、通信時間、同期時間の平均を出す。

通信遅延 L、コスト g を(L=16, g=4) (L=16, g=16) (L=64, g=4) (L=64g=16) というパターンの時に、データ数 n が 128、256、512、1024 の 4 つの場合に対してプロセッサ数を 2、4、8、16 と変化させたときの内部計算時間、通信時間、同期時間を測定し総実行時間を出す。

3.3　実験方法

付録に本実験で作成したプログラムを示す。付録に示したプログラムをファイルを用いて、BSP 上での実行時間である内部計算時間、通信時間、同期時間、総実行時間計測する。

3.4　シミュレーションプログラム

本節では付録に示したシミュレートプログラムについて説明する。シミュレートプログラムは以下のクラスファイルから成る。

・BspSelection.class

・BspProcessor.class

・BspNetwork.class

・Makedata.class

BspSelection は通信コスト、通信遅延、データ数、プロセッサ数を調査したい数値に任意に変更できる。また、BspProcessor、BspNetwork での実行処理の時間計算、実行時間を出す。

BspNetwork はプロセッサ p 台を結ぶネットワークをつくる。

BspProcessor は Selection アルゴリズムをプログラムで実装している。

Makedata は Inputdata 選択する数字の列を事前に準備しており、そこからランダムに求めるべきデータ数を抽出できるようになっている。また実行時間出力を Outputdata に出力する。

第４章　結　果

以下に本研究で作成したシミュレートプログラムによる選択問題の処理時間の測定結果を示す

通信遅延 L、コスト g を(L=16, g=4) (L=16, g=16) (L=64, g=4) (L=64g=16) というパターンの時に、データ数 n が 128、256、512、1024 の 4 つの場合に対してプロセッサ数を 2、4、8、16 と変化させたときの内部計算時間、通信時間、同期時間各 20 回の平均を測定し、これを表 1 ~16 に示す。

表 1：n=128 のときの内部計算時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	1324	1267	1404	1467
4	1139	1184	1120	1156
8	1154	1261	1210	1215
16	1607	1587	1623	1621

表 2：n=128 のときの通信時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	1076	4368	1100	4432
4	1344	5472	1364	5504
8	1840	7504	1864	7520
16	2608	10448	2608	10432

表 3：n=128 のときの同期時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	80	80	320	320
4	144	144	576	576
8	208	208	832	832
16	272	272	1088	1088

表 4：n=128 のときの総実行時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	2483	5718	2824	6233
4	2627	6808	3063	7249
8	3204	8979	3909	9580
16	4489	12308	5321	13152

表 5：n=256 のときの内部計算時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	2662	2624	2652	2583
4	1908	1843	2076	1872
8	1508	1665	1448	1544
16	1813	1823	1860	1795

表 6：n=256 のときの通信時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	1652	6912	1712	6960
4	1628	6320	1684	6640
8	1972	8064	1940	7920
16	2660	10640	2680	10624

表 7：n=256 のときの同期時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	80	80	320	320
4	144	144	576	576
8	208	208	832	832
16	272	272	1088	1088

表 8：n=256 のときの総計算時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	2483	5718	2824	6233
4	2627	6808	3063	7249
8	3204	8979	3909	9580
16	4489	12308	5321	13152

表 9：n=512 のときの内部計算時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	5155	5635	5112	5151
4	3192	3241	3569	3498
8	2238	2275	2496	2461
16	2213	2232	2317	2238

表 10：n=512 のときの通信時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	2892	12176	2956	11360
4	2052	8672	2388	8640
8	2180	8896	2244	8976
16	2768	11088	2820	11120

表 11：n=512 のときの同期時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	80	80	320	320
4	144	144	576	576
8	208	208	832	832
16	272	272	1088	1088

表 12：n=512 のときの総計算時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	8127	17891	8388	16831
4	5388	12057	6533	12714
8	4626	11379	5572	12269
16	5253	13592	6225	14446

表 13：n=1024 のときの内部計算時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	11604	10203	11068	10512
4	5763	6499	6488	6785
8	4391	3897	3549	3735
16	3009	3098	2887	2992

表 14：n=1024 のときの通信時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	5352	20928	5716	20608
4	3232	13504	3216	12816
8	2792	10752	2584	10672
16	2972	12096	2948	11968

表 15：n=1024 のときの同期時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	80	80	320	320
4	144	144	576	576
8	208	208	832	832
16	272	272	1088	1088

表 16：n=1024 のときの総計算時間

p	L=16,g=4	L=16,g=16	L=64,g=4	L=64,g=16
2	17038	31215	17107	31444
4	9140	20161	10282	20180
8	7394	14864	6966	15244
16	6255	15244	6925	16048

表 1～表 16 より、横軸にプロセッサ数、縦軸に総計算時間をデータ数別に表したグラフを図 1、2、3、4 に示す

図 1：n=128 のときの総計算時間とプロセッサ台数の関係

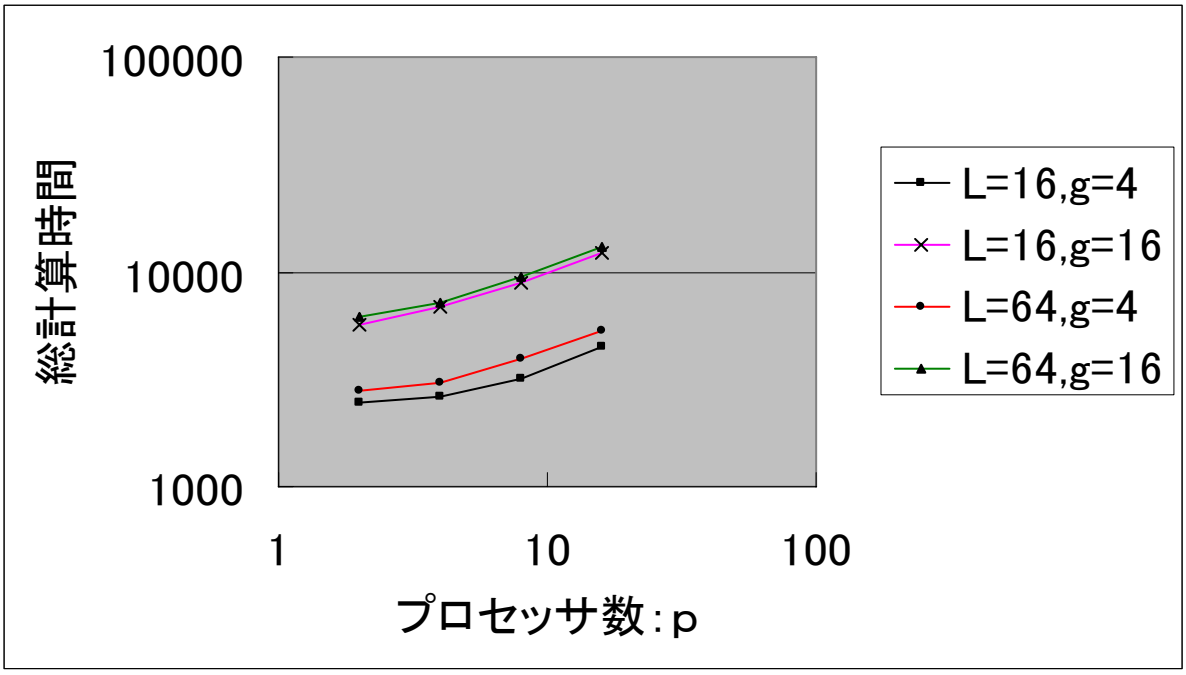


図 2：n=256 のときの総計算時間とプロセッサ台数の関係

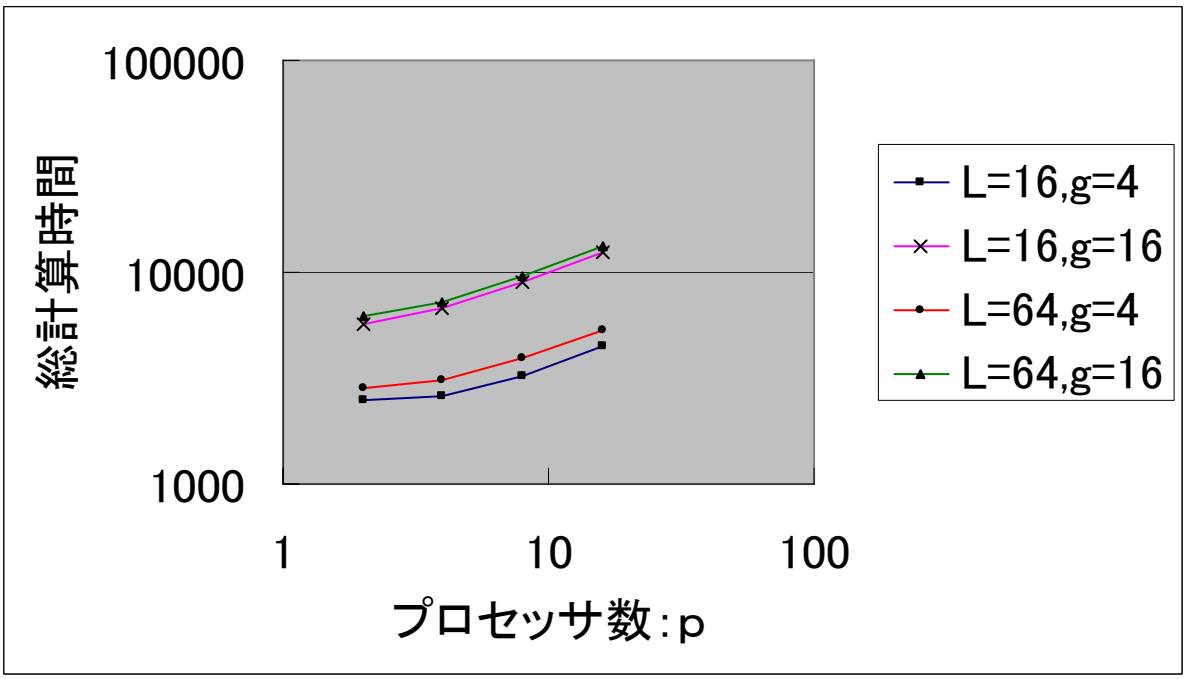


図 3：n=512 のときの総計算時間とプロセッサ台数の関係

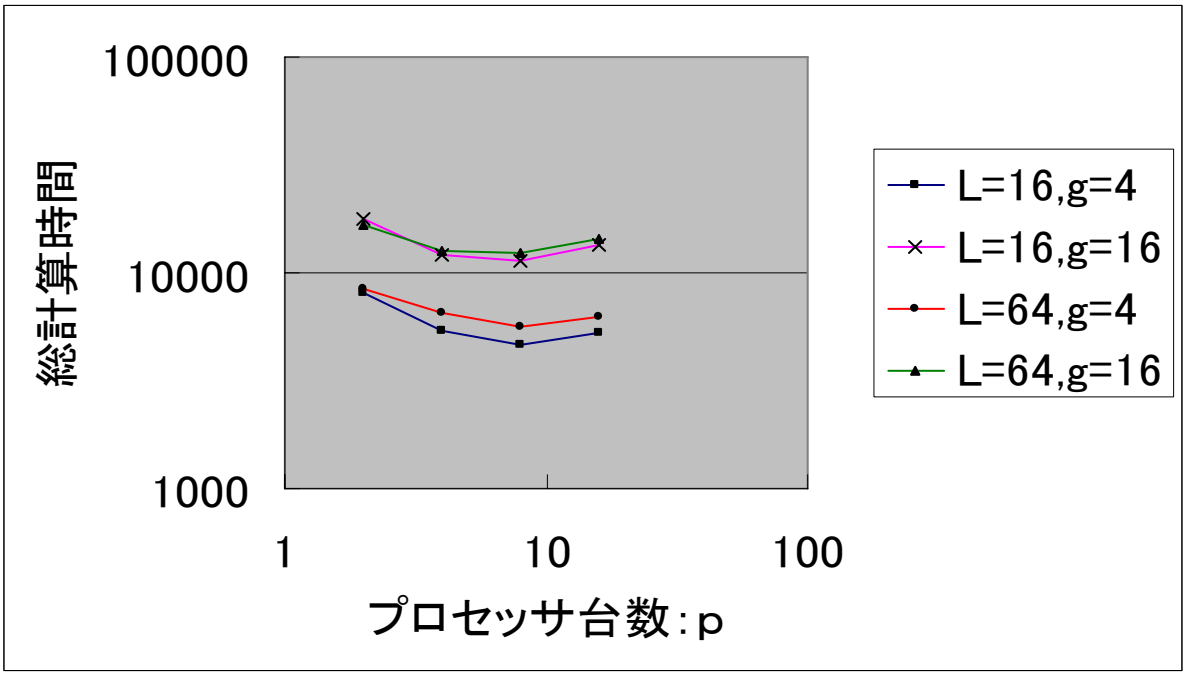
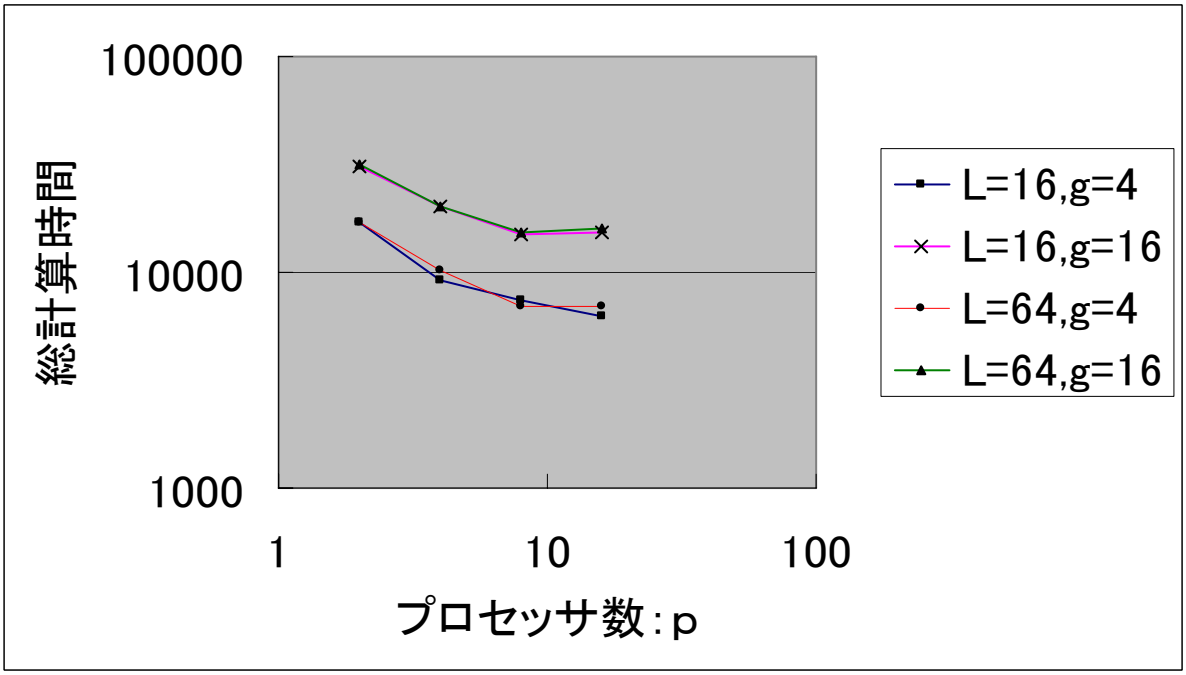


図 4：n=1024 のときの総計算時間とプロセッサ台数の関係



第５章　考　察

図よりデータ数が少ない場合、プロセッサ台数をある少なくしたほうが実行時間を短縮できることがわかる。

表よりプロセッサ数を増やすことにより、プロセッサ 1 台辺りの処理データが減るため内部計算時間が減少する一方、プロセッサ内の同期を取る回数が増える。このため、内部計算時間に対して同期時間が大きくなるようなプロセッサ台数のときはプロセッサ数を増やすと全体の時間が長くなると考えられる。通信時間と同期時間との関係についても同様のことが言える。よって、効率的にアルゴリズムを実行するためにはデータ数に対しての最適なプロセッサ数を求める必要がある。

表より、通信コスト g が増加した場合、1 回の通信時間に対して比例して増加し、また通信延期 L も同様に一回の同期時間に対して比例的に増加する。L,g が増加したとしても内部計算時間は変わることはない。

よって、通信コスト g、通信延期 L を抑えることにより、より速い総実行時間を出すことが可能である。つまり、より高速な通信網を使用することにより通信コストを抑えることができる。通信コストが抑えられない場合に対しては通信命令を抑えたアルゴリズムを考えることが今後の課題となる。

第６章　結　論

本研究で選択問題を解く並列アルゴリズムの実行をシミュレートするプログラムを作成し、BSP モデル上で選択問題を解くアルゴリズムの正当性および計算量の実験的評価を行なった。

対象となるデータのサイズが小さい時はプロセッサ数が増加するとかえって全体の実行時間が長くなってしまう。従って内部計算時間、通信時間を抑え、データ数に対するより的確なプロセッサ台数を提案するアルゴリズムを考えることが今後の課題となる。

謝辞

本研究するにあたり、様々な助言、温かい御指導をしていただいた近畿大学理工学部情報学科 石水隆先生には心より感謝申し上げます。
また、研究室の皆様には論文を書く上で助言等をしてくださり大変感謝いたします。

参考文献

[1] 石水隆 他，選択問題を解く BSP モデルおよび BSP*モデル上の並列アルゴリズム，信学技報, 1999.

[2] J.JàJà, "An Introduction to Parallel Algorithms," Addison - Wesley Publishing Company, 1999

[3] L.G.Valiant, "A Bridging Model for Parallel Computation, ", Communications of the ACM, Vol.33, No.8, pp.103--111, 1990

付録

以下に本研究で作成した BSP モデル上の選択問題を解いた並列アルゴリズムのシミュレートプログラムを示す。

(1) inputFile.java

(2) BspSelection.java

(3) BspNetwork.java

(4) BspProcessor.java

```
/* InputFile.java  */


import ioTools.*;                // ファイル入出力用
import java.io.*;                // ファイル入出力用
import java.util.*;              // 文字列処理用

public class InputData {
        final int range = 100;                  // データの範囲
        int[] array;                            // データ

        public InputData (int n) {
                array = new int [n];
        }

        // 要素数 array.length の 0～range-1 までのランダムデータを作成する
        public void makeRandomData () {
                for (int i=0; i<array.length; i++)
                        array[i] = (int) (Math.random() * range);
        }

        // 要素数 array.length のソート済みデータを作成する
        public void makeSortedData () {
                for (int i=0; i<array.length; i++)
                        array[i] = (int) (i*range/array.length);
        }

        // 全てのデータを得る
        public int[] get () {
                return array;
        }
```

```
// array[low]～array[high]までのデータを得る
public int[] get (int low, int high) {
	if (low < 0 || high >= array.length || low > high)
		return null;
	else {
		int[] a = new int[high - low +1];
		for (int i=0; i<a.length; i++)
			a[i] = array[low + i];
		return a;
	}
}

// データを表示する
public void dump () {
	for (int i=0; i<array.length; i++)
		System.out.print (array[i] + " ");
	System.out.println();
}

// データをデフォルトファイルに出力する
public void dumpToFile () {
	dumpToFile ("InputData.txt");
}

// データを指定したファイルに出力する
public void dumpToFile (String fileName) {
	PrintWriter outputFile = FileIo.fWrite(fileName, false);
	for (int i=0; i<array.length; i++)
		outputFile.print (array[i] + " ");
	outputFile.println();
	outputFile.close();
}

// デフォルトファイルからデータを読み込む
public void readData () {
	readData ("InputData.txt");
}

// 指定したファイルからデータを読み込む
public void readData(String fileName) {
	BufferedReader buffer = FileIo.fRead(fileName);
	String line = readLine(buffer);
	StringTokenizer st = new StringTokenizer(line);
	int arraySize = st.countTokens();
	for (int i=0; i<arraySize; i++)
		array[i] = Integer.parseInt(st.nextToken());
}
```

```
        // ファイルから 1 行読み込む
        String readLine(BufferedReader buffer) {
                String line = "";
                try {
                        line = buffer.readLine();
                } catch(IOException error_report) {
               /* 読込みエラーが発生したら, キャッチした例外を表示し,
                   ファイルを閉じ, 処理系を終了させる */
                        System.out.println(error_report);
                        System.exit(1);
                }
                return line;
        }

        public static void main (String[] args) {
                System.out.print("size : ");
                int n = Console.ReadInteger();
                InputData data = new InputData(n);
                data.makeRandomData();
                data.dump ();
                data.dumpToFile();
        }
    }
```

/* InputFile.java ここまで */

/* BspSelection.java */

```
import ioTools.*;  //ファイル入出力用
import java.io.*;  //ファイル入出力用

public class BspSelection {



    static int gap = 4;                    // 通信コスト
    static int latency = 16;               // 通信遅延
    static int numOfProcessors = 64;       // プロセッサ数
    static int numOfData = 1024;           // データ数

    static BspProcessor[] processor;       // プロセッサ
```

```
static BspNetwork network;                    // ネットワーク
static InputData input;                       // 入力データ

static boolean debugSW = false;               // デバグ用スイッチ
static boolean traceSW = false;               // トレス用スイッチ

static PrintWriter output;                    // 出力用ファイル
static PrintWriter log;                       // ログ出力用ファイル


public static void main (String[] args) {

        // ネットワークを作る
        network = new BspNetwork (numOfProcessors);

        // プロセッサを作る
        processor = new BspProcessor[numOfProcessors];
        for (int p=0; p<numOfProcessors; p++) {          // プロセッサ i を作る
                processor[p] = new BspProcessor (p, numOfProcessors);
                processor[p].setNetwork (network);
        }

        // 出力用ファイル設定
        output = FileIo.fWrite ("OutputData.txt", false);

        log = FileIo.fWrite ("BspSelection.log", false);
        network.setLogFile (log);
        for (int p=0; p<numOfProcessors; p++) {
                processor[p].setOutputFile (output);
                processor[p].setLogFile (log);
        }

        for (numOfData = 128; numOfData <= 1024; numOfData *= 2) {
                for (numOfProcessors = 2; numOfProcessors <=16; // && numOfProcessors
<=numOfData/16;
                        numOfProcessors *=2) {
                        for (latency = 16; latency <=64; latency *=4) {
                                for (gap = 4; gap <=16; gap *=4) {
```

```
                                        // 時計初期化
                                        for (int i=0; i<numOfProcessors; i++)
                                                processor[i].setTime (0,0,0);  // 各時間を 0 で初期化す
る

                                        for (int i=0; i<20; i++) {
                                        // ランダムデータ作成
                                                input = new InputData (numOfData);
                                                input.makeRandomData(); // ソート済データが必要な
ら input.makeSortedData() を用いる
                                                if (traceSW) input.dump();
                                                input.dumpToFile();

                                                // 各プロセッサが保持する初期値設定
                                                for (int p=0; p<numOfProcessors; p++) {
                                                        int low = p * numOfData / numOfProcessors;
                                                        int high = (p+1) * numOfData /
numOfProcessors - 1;
                                                        processor[p].setData (input.get (low, high));
                                                }

                                                // ランク入力（プロセッサ 0 が保持）
                                                int rank = numOfData/2;
                                                processor[0].setRank（rank);  // プロセッサ 0 にラン
クを与える

/*
                        // 初期データ表示
                        System.out.println ("Input Data");
                        for (int p=0; p<numOfProcessors; p++) {
                                processor[p].showData();
                                processor[p].logData();

*/

                                                // 選択
                                                parallelSelection ();    // 並列選択
                                                sequentialSelection (); // 逐次選択
/*
```

```
                    // 目的データ表示
                    processor[0].showTarget();
                    processor[0].printTarget();
                    processor[0].logTarget();
*/
                                                }

                                                // 実行時間表示
                                                showTime();
                                                printTime();
                                                logTime();
                                        }
                                }
                        }
                }
                // 出力用ファイルを閉じる
                output.close();
                log.close();
        }

        // 並列選択
        public static void parallelSelection () {
                for (int i=1; i<numOfProcessors; i*=2) { // log p 回繰り返す
                        // 基準値候補選択
                        if (debugSW) System.out.println ("selectPivotCandidate");
                        log.println ("selectPivotCandidate");
                        for (int p=0; p<numOfProcessors; p++)
                                processor[p].selectPivotCandidate();

                        // 基準値候補をプロセッサ 0 に送信
                        if (debugSW) System.out.println ("sendPivotCandidate");
                        log.println ("sendPivotCandidate");
                        for (int p=0; p<numOfProcessors; p++)
                                processor[p].sendPivotCandidate();

                        synchronous(); // 同期

                        // 基準値候補受信
```

```
if (debugSW) System.out.println ("receivePivotCandidate");
log.println ("receivePivotCandidate");
processor[0].receivePivotCandidate();

// 基準値選択
if (debugSW) System.out.println ("selectPivot");
log.println("selectPivot");
processor[0].selectPivot();

// 全てのプロセッサに基準値送信
if (debugSW) System.out.println ("sendPivot");
log.println("sendPivot");
        processor[0].broadcastPivot ();

synchronous(); // 同期

// 基準値受信
if (debugSW) System.out.println ("receivePivot");
log.println ("receivePivot");
for (int p=1; p<numOfProcessors; p++)
        processor[p].receivePivot();

// 保持するデータを基準値未満の部分と以上の部分に分割する
if (debugSW) System.out.println ("devideData");
log.println ("devideData");
for (int p=0; p<numOfProcessors; p++)
        processor[p].devideData();
if (traceSW)                    // データの表示
        for (int p=0; p<numOfProcessors; p++)
                processor[p].showData();
for (int p=0; p<numOfProcessors; p++)
        processor[p].logData();

// 基準値未満のデータ数をプロセッサ 0 に送信する
if (debugSW) System.out.println ("sendNumOfLowData");
log.println ("sendNumOfLowData");
for (int p=0; p<numOfProcessors; p++)
        processor[p].sendNumOfLowData();
```

```
synchronous(); // 同期

// 基準値未満のデータ数を受信する
if (debugSW) System.out.println ("receiveNumOfLowData");
log.println ("receiveNumOfLowData");
processor[0].receiveNumOfLowData();

// 基準値未満のデータ数の和を求める
if (debugSW) System.out.println ("sumNumOfAllLowData");
log.println ("sumNumOfAllLowData");
processor[0].sumNumOfAllLowData();
if (traceSW) {
        processor[0].showRank();
}
processor[0].logRank();

// 目的のデータがどちらの部分データに存在するか判定する
if (debugSW) System.out.println ("selectPartOfData");
log.println ("selectPartOfData");
processor[0].selectPartOfData();

// 目的のデータがどちらの部分データに存在するかを他のプロセッサに伝える
if (debugSW) System.out.println ("broadcastWhereIsTarget");
log.println ("broadcastWhereIsTarget");
processor[0].broadcastWhereIsTarget();

synchronous(); // 同期

// 目的のデータがどちらの部分データに存在するかを受信する
if (debugSW) System.out.println ("receiveWhereIsTarget");
log.println ("receiveWhereIsTarget");
for (int p=1; p<numOfProcessors; p++)
        processor[p].receiveWhereIsTarget();

// 目的のデータが含まれる部分データを選ぶ
if (debugSW) System.out.println ("selectSubData");
log.println ("selectSubData");
for (int p=0; p<numOfProcessors; p++)
        processor[p].selectSubData();
```

```
                if (traceSW)
                        for (int p=0; p<numOfProcessors; p++)
                                processor[p].showData();
                for (int p=0; p<numOfProcessors; p++)
                        processor[p].logData();
        }
}


// 逐次選択
public static void sequentialSelection () {
        // 全てのデータをプロセッサ 0 に送信する
        if (debugSW) System.out.println ("sendData");
        log.println ("sendData");
        for (int p=0; p<numOfProcessors; p++)
                processor[p].sendData();

        synchronous(); // 同期

        // データを受信する
        if (debugSW) System.out.println ("receiveData");
        log.println ("receiveData");
        processor[0].receiveData();

        // プロセッサ 0 で逐次に選択を行う
        if (debugSW) System.out.println ("sequentialSelection");
        log.println ("sequentialSelection");
        processor[0].sequentialSelection();
}

// 全てのプロセッサ間の同期
public static void synchronous () {
        synchronous (0, numOfProcessors-1);
}

// プロセッサ p～プロセッサ q 間の同期
public static void synchronous (int p, int q) {
        if (traceSW)
                for (int i=p; i<=q; i++)
```

```
                        network.showSendQueue (i);
        for (int i=p; i<=q; i++)
                network.logSendQueue (i);

        // ネットワークの送信キュー内のデータを受信キューに移す
        network.synchronous (p, q);

        int timeI = 0;
        int timeG = 0;
        int timeL = 0;
        for (int i=p; i<=q; i++) { // プロセッサ p～プロセッサ q の時間を最大値に揃える
                if (processor[i].timeI > timeI) timeI = processor[i].timeI;
                if (processor[i].timeG > timeG) timeG = processor[i].timeG;
                if (processor[i].timeL > timeL) timeL = processor[i].timeL;
        }

        timeL++;

        for (int i=p; i<=q; i++)
                processor[i].setTime (timeI, timeG, timeL);
}



// 実行時間表示
static void showTime() {
        System.out.println("p = " + numOfProcessors);
        System.out.println("n = " + numOfData);
        System.out.println("L = " + latency);
        System.out.println("g = " + gap);
        System.out.println("time : " + (processor[0].timeI/20) + " + g*" + (processor[0].timeG/20) + " + L*"
+ (processor[0].timeL/20)
        + " Total: "+ ((processor[0].timeI + processor[0].timeG*gap + processor[0].timeL*latency)/20));
}



// 実行時間出力
static void printTime() {
```

```
            output.println("p = " + numOfProcessors);
            output.println("n = " + numOfData);
            output.println("L = " + latency);
            output.println("g = " + gap);
            output.println("time : " + (processor[0].timeI/20) + " + g*" + (processor[0].timeG/20) + " + L*" +
    (processor[0].timeL/20)
            + " Total: "+ ((processor[0].timeI + processor[0].timeG*gap + processor[0].timeL*latency)/20));
    }

    // 実行時間出力(ログ出力用)
    static void logTime() {
            log.println("p = " + numOfProcessors);
            log.println("n = " + numOfData);
            log.println("L = " + latency);
            log.println("g = " + gap);
            log.println("time : " + (processor[0].timeI/20) + " + g*" + (processor[0].timeG/20) + " + L*" +
    (processor[0].timeL/20)
            + " Total: "+ ((processor[0].timeI + processor[0].timeG*gap + processor[0].timeL*latency)/20));



}
}
```

/* BspSelection.java ここまで */

/* BspNetwork.java */

```
import java.util.ArrayList;  // ArrayList 処理用
import ioTools.*;  //ファイル入出力用
import java.io.*;  //ファイル入出力用

public class BspNetwork {
    ArrayList[] sendQueue;          // プロセッサ i へ送信中のデータ
    ArrayList[] receiveQueue;       // プロセッサ i が受信中のデータ

    PrintWriter log;                // ログ出力用ファイル

    // プロセッサ p 台を結ぶネットワークを作る
    public BspNetwork (int p) {
```

```
        sendQueue = new ArrayList[p];
        receiveQueue = new ArrayList[p];
        for (int i=0; i<p; i++) {
                sendQueue[i] = new ArrayList();
                receiveQueue[i] = new ArrayList();
        }
}

// int 型データ n をプロセッサ p への送信キューに置く
public void put (int p, int n) {
        sendQueue[p].add (new Integer(n));
}

// int 型配列データ a をプロセッサ p への送信キューに置く
public void putArray (int p, int[] a) {
        for (int i=0; i<a.length; i++)
                sendQueue[p].add (new Integer(a[i]));
}

// 2 個組 int 型データ(m,n)をプロセッサ p への送信キューに置く
public void putPair(int p, int m, int n) {
        sendQueue[p].add (new Integer(m));
        sendQueue[p].add (new Integer(n));
}

// int 型配列データ a の a[l]～a[h]をプロセッサ p への送信キューに置く
// l,h が 0 未満または a.length 以上のときはエラー
public void putPartOfArray (int p, int[]a, int l, int h) {
        if (l<0 || l>=a.length || h<0 || h>=a.length)
                executeError("Illegal index of array at Queue "+p);
        for (int i=l; i<=h; i++) {
                sendQueue[p].add (new Integer(a[i]));
        }
}

// int 型行列データ m をプロセッサ p への送信キューに置く
public void putMatrix (int p, int[][]m) {
        for (int i=0; i<m.length; i++)
                for (int j=0; j<m[i].length; j++)
```

```
                        sendQueue[p].add (new Integer(m[i][j]));
}

// boolean 型データ b をプロセッサ p への送信キューに置く
public void putBool (int p, boolean b) {
        sendQueue[p].add (Boolean.valueOf (b));
}

// プロセッサ p への受信キューから int 型データを取り出す
// 受信キューにデータが入っていない場合、取り出したデータが Integer 型でない場合はエラー
public int get (int p) {
        if (receiveQueue[p].isEmpty()) executeError("Queue "+p+" Underflow");
        if ((receiveQueue[p].get(0)).getClass() !=  Integer.class) executeError("Queue "+p+" Type
Mismatched");
        int n = ((Integer) receiveQueue[p].get(0)).intValue();
        receiveQueue[p].remove(0);
        return n;
}

// プロセッサ p への受信キューから int 型配列データを取り出す
// 受信キューにデータが入っていない場合、取り出したデータが Integer 型でない場合はエラー
public int[] getArray (int p) {
        if (receiveQueue[p].isEmpty()) executeError("Queue "+p+" Underflow");
        int[] a = new int[receiveQueue[p].size()];
        for (int i=0; i<a.length; i++) {
                if  ((receiveQueue[p].get(0)).getClass()  !=   Integer.class)  executeError("Queue
"+p+" Type Mismatched");
                a[i] = ((Integer) receiveQueue[p].get(0)).intValue();
                receiveQueue[p].remove(0);
        }
        return a;
}

// プロセッサ p への受信キューから 2 個組 int 型データを取り出す
// 受信キューに 2 個以上データが入っていない場合、取り出したデータが Integer 型でない場合はエラー
public int[] getPair (int p) {
        if (receiveQueue[p].size() < 2) executeError("Queue "+p+" Underflow");
        if ((receiveQueue[p].get(0)).getClass() !=  Integer.class) executeError("Queue "+p+" Type
Mismatched");
```

```
        int m = ((Integer) receiveQueue[p].get(0)).intValue();
        receiveQueue[p].remove(0);
        if ((receiveQueue[p].get(0)).getClass() !=  Integer.class) executeError("Queue "+p+" Type Mismatched");
        int n = ((Integer) receiveQueue[p].get(0)).intValue();
        receiveQueue[p].remove(0);
        int[] intPair = {m,n};
        return intPair;
}

// プロセッサ p への受信キューから長さ s の int 型配列データを取り出す
// 受信キューに s 個以上データが入っていない場合、取り出したデータが Integer 型でない場合はエラー
public int[] getArray (int p, int s) {
        if (receiveQueue[p].size() < s) executeError("Queue "+p+" Underflow");
        int[] a = new int[s];
        for (int i=0; i<s; i++) {
                if ((receiveQueue[p].get(0)).getClass() != Integer.class) executeError("Queue "+p+" Type Mismatched");
                a[i] = ((Integer) receiveQueue[p].get(0)).intValue();
                receiveQueue[p].remove(0);
        }
        return a;
}

// プロセッサ p への受信キューからサイズ s×s の int 型行列データを取り出す
// 受信キューに s*s 個以上データが入っていない場合、取り出したデータが Integer 型でない場合はエラー
public int[][] getMatrix (int p, int s) {
        if (receiveQueue[p].size() < s*s) executeError("Queue "+p+" Underflow");
        int[][] m = new int[s][s];
        for (int i=0; i<s; i++)
                for (int j=0; j<s; j++) {
                        if ((receiveQueue[p].get(0)).getClass() != Integer.class) executeError("Queue "+p+" Type Mismatched");
                        m[i][j] = ((Integer) receiveQueue[p].get(0)).intValue();
                        receiveQueue[p].remove(0);
        }
        return m;
}
```

```
// プロセッサ p への受信キューからサイズ s×t の int 型行列データを取り出す
// 受信キューに s*t 個以上データが入っていない場合、取り出したデータが Integer 型でない場合はエラー
public int[][] getMatrix (int p, int s, int t) {
        if (receiveQueue[p].size() < s*s) executeError("Queue "+p+" Underflow");
        int[][] m = new int[s][t];
        for (int i=0; i<s; i++)
                for (int j=0; j<t; j++) {
                        if     ((receiveQueue[p].get(0)).getClass()     !=          Integer.class)
executeError("Queue "+p+" Type Mismatched");
                        m[i][j] = ((Integer) receiveQueue[p].get(0)).intValue();
                        receiveQueue[p].remove(0);
        }
        return m;
}

// プロセッサ p への受信キューから boolean 型データを取り出す
// 受信キューにデータが入っていない場合、取り出したデータが Boolean 型でない場合はエラー
public boolean getBool (int p) {
        if (receiveQueue[p].isEmpty()) executeError("Queue "+p+" Underflow");
        if ((receiveQueue[p].get(0)).getClass() !=  Boolean.class) executeError("Queue "+p+" Type
Mismatched");
        boolean b = ((Boolean) receiveQueue[p].get(0)).booleanValue();
        receiveQueue[p].remove(0);
        return b;
}

// プロセッサ p～プロセッサ q の送信キューのデータを受信キューに移す
// (この操作を行わないと受信キューからデータを取り出せない)
public void synchronous(int p, int q) {
        for (int i=p; i<=q; i++) {
                while (!(sendQueue[i].isEmpty())) { // 送信キューが空になるまで
                        receiveQueue[i].add(sendQueue[i].get(0));
                        sendQueue[i].remove(0);
                }
        }
}

// プロセッサ p への受信キューをクリアする
public void clear (int p) {
```

```
        receiveQueue[p].clear();
}

// プロセッサ p への受信キューが空であるか？
public boolean isEmpty (int p) {
        return receiveQueue[p].isEmpty();
}

// プロセッサ p への受信キューに置かれたデータの数を返す
public int size (int p) {
        return receiveQueue[p].size();
}

// ログ出力用ファイルをセットする
public void setLogFile(PrintWriter l) {
        log = l;
}

// プロセッサ p への送信キューの内容を表示する(デバグ用)
public void showSendQueue (int p) {
        if (p < 10)
                System.out.print ("Pr. "+p+":");
        else System.out.print ("Pr."+p+":");
        System.out.println (formatQueue (sendQueue[p]));
}

// プロセッサ p への受信キューを表示する(デバグ用)
public void showReceiveQueue (int p) {
        if (p < 10)
                System.out.print ("Pr. "+p+":");
        else System.out.print ("Pr."+p+":");
        System.out.println (formatQueue (receiveQueue[p]));
}

// プロセッサ p への送信キューをログファイルに出力する
public void logSendQueue (int p) {
        if (p < 10)
                log.print ("Pr. "+p+":");
        else log.print ("Pr."+p+":");
```

```
        log.println (formatQueue (sendQueue[p]));
}

// プロセッサ p の受信キューをログファイルに出力する
public void logReceiveQueue (int p) {
        if (p < 10)
                log.print ("Pr. "+p+":");
        else log.print ("Pr."+p+":");
        log.println (formatQueue (receiveQueue[p]));
}

// キューを出力用に整形する
String formatQueue (ArrayList queue) {
        if (queue.isEmpty())
                return " Empty";
        else {
                String str = "";
                for (int i=0; i<queue.size(); i++)
                        str += formatData (queue.get(i));
                return str;
        }
}

// データを出力用に整形する
String formatData (Object o) {
        if (o.getClass() ==  Integer.class) {
                int val = ((Integer) o).intValue();
                if (val == Integer.MAX_VALUE) return " --";       // 無限大は"--"を出力
                else if (val < 10) return "  "+val;
                else return " "+val;
        } else if (o.getClass() == Boolean.class) {
                if (((Boolean) o).booleanValue()) return "  T"; else return "  F";
        } else return " ??";
}

static void executeError (String err_mes) { /* 実行時エラー */
        System.out.println("Execute error");
        System.out.println(err_mes);
        System.exit(1);
```

```
    }
}
```

/* BspNetwork.java ここまで　*/

/* BspProcessor.java　*/

```
import ioTools.*;  //ファイル入出力用
import java.io.*;  //ファイル入出力用

public class BspProcessor {
    int processorNumber;        // プロセッサ番号
    BspNetwork network;         // ネットワーク
    int numOfProcessors;        // プロセッサ数
    int timeI;                  // 内部計算時間
    int timeG;                  // 通信時間
    int timeL;                  // 同期回数
    int[] data;                 // データ配列

    int pivot;                  // 基準値
    int[] pivotCandidate;       // 基準値候補

    int rank;                   // ランク（プロセッサ 0 のみが使用）
    int target;                 // 目的データ（プロセッサ 0 のみが使用）

    int lowLeft;                // データ基準値以下の部分の左端
    int lowRight;               // データ基準値以下の部分の右端
    int highLeft;               // データ基準値以上の部分の左端
    int highRight;              // データ基準値以上の部分の右端
    int numOfLowData;           // データ基準値以下の部分のデータ数
    int numOfHighData;          // データ基準値以上の部分のデータ数
    int[] numOfAllLowData;      // 全てのプロセッサでのデータ基準値以下の部分のデータ数（プロセッサ 0 のみが使用）
    boolean dataIsInLeftPart; // 目的データがどちらの部分データに存在するか

    PrintWriter output;         // 出力用ファイル
    PrintWriter log;            // ログ出力用ファイル
```

```
// コンストラクタ
// プロセッサ番号とプロセッサ台数を設定する
public BspProcessor(int p, int np) {
        processorNumber = p;
        numOfProcessors = np;
}

// ネットワークを設定する
public void setNetwork (BspNetwork net) {
        network = net;
}

// データをセットする
public void setData (int[] a) {
        data = a;
        rank = -1;                          // 初期値は便宜上-1 を入れる
        target = Integer.MAX_VALUE; // 初期値は便宜上無限大を入れる
}

// 時間をセット
public void setTime (int i, int g, int l) {
        timeI = i;
        timeG = g;
        timeL = l;
}

// 出力用ファイルをセット
public void setOutputFile(PrintWriter o) {
        output = o;
}

// ログ出力用ファイルをセット
public void setLogFile(PrintWriter l) {
        log = l;
}

// ランクをセット
// このメソッドはプロセッサ 0 のみが使用する
public void setRank (int r) {
```

```
        rank = r;
}

// 基準値候補選択
public void selectPivotCandidate() {
        if (data == null) {
                pivot = Integer.MAX_VALUE; // データを持たない場合は便宜上最大値を入れる
                timeI++;
        } else {
                pivot = data[(int) (Math.random()*data.length)];
                // ここでは基準値をデータからランダムに選んでいるが、
                // pivot = selectMedian (data);
                // pivot = semiSelectMedian (data)
                // pivot = sampleMedian (data, 3)
                // のいずれかを用いても良い
                // (どれが速いかはデータによる)
                timeI++;            // pivot=
        }
        timeI++;   // if
}

// プロセッサ 0 に基準値候補送信
public void sendPivotCandidate() {
        if (pivot != Integer.MAX_VALUE)
                send (0, pivot);
        timeI++;   // if
}

// 基準値候補受信
// このメソッドはプロセッサ 0 のみが実行する
public void receivePivotCandidate() {
        pivotCandidate = receiveArray ();
        clearNetwork ();
}

// 基準値選択
// このメソッドはプロセッサ 0 のみが実行する
public void selectPivot() {
        if (pivotCandidate == null) {   /* 基準値候補が存在しないときは */
```

```
                pivot = Integer.MAX_VALUE; /*  便器上無限大を与える  */
                timeI++;    // pivot=
        } else {
                pivot = selectMedian (pivotCandidate);
                //  ここでは select メソッドを用いて基準値候補の中央値を選んでいるが、
                // pivot = pivotCandidate[(int) (Math.random()*pivotCandidate.length)];
                // pivot = selectMedian (pivotCandidate);
                // pivot = sampleMedian (pivotCandidata, 3);
                //  のいずれかを用いても良い
                // (どれが速いかはデータによる)
                timeI++;    // pivot=
        }
         timeI++;    // if
}

// 全てのプロセッサに基準値送信
// このメソッドはプロセッサ 0 のみが実行する
public void broadcastPivot() {
        if (pivot != Integer.MAX_VALUE) {
                for (int i=1; i<numOfProcessors; i++)
                        send (i, pivot);
                timeI += numOfProcessors-1; // for
        }
        timeI++;       // if
}

// 基準値受信
public void receivePivot() {
        pivot = receive ();
        clearNetwork ();
}

// データ配列 data を基準値 pivot 未満の部分と以上の部分に分割し、
// lowLeft, lowRight, highLeft, highright の値を設定する
// 実行後は lowRight+1 == highLeft または lowRight+2 == highLeft, lowLeft == 0, highRight ==
data.length-1 となっており
// a[lowLeft]～a[lowRight]に piv 以下の値、a[highLeft]～a[highRight]に pivot 以上のが入っている
// lowRight+2 == highLeft とのきは、a[lowRight+1] == pivot となっている
// また、基準値未満のデータ数を numOfLowData に、基準値以上のデータ数を numOfHighData が入る
```

```
public void devideData() {
        if (data == null) {   /* データが存在しない場合*/
                lowLeft = Integer.MAX_VALUE;   /* 各変数の値を便器上無限大にする */
                lowRight = Integer.MIN_VALUE;
                highLeft = Integer.MAX_VALUE;
                highRight = Integer.MIN_VALUE;
                numOfLowData = 0;
                numOfHighData = 0;
                timeI += 6;        // lowLeft=, lowRight=, highLeft=, highRight=, numOfLowData=,
numOfHighData=
        } else {
                divide (data, 0, data.length-1, pivot);
                numOfLowData = lowRight+1;
                numOfHighData = highRight - lowRight;
                timeI += 2;      // numOfLowData=, numOfHighData=
        }
        timeI++;
}

// 基準値以下のデータ数をプロセッサ 0 に送信する
public void sendNumOfLowData() {
        send (0, numOfLowData);
}

// 基準値未満のデータ数を受信する
// このメソッドはプロセッサ 0 のみが実行する
public void receiveNumOfLowData() {
        numOfAllLowData = receiveArray ();
        clearNetwork ();
}

// 基準値未満のデータ数の和を求める
// このメソッドはプロセッサ 0 のみが実行する
public void sumNumOfAllLowData() {
        for (int i=1; i<numOfAllLowData.length; i++)
                numOfAllLowData[0] += numOfAllLowData[i];
        timeI += (numOfAllLowData.length-1)*2;   // for, numOfAllLowData[]=
}
```

```
// 目的のデータがどちらの部分データに存在するか判定する
// 目的のデータが基準値未満の部分データに存在するときは dataIsInLeftPart を true に、
// 存在しないときは false にする
// このメソッドはプロセッサ 0 のみが実行する
public void selectPartOfData() {
        dataIsInLeftPart = (numOfAllLowData[0] > rank);
        if (!dataIsInLeftPart) {
                rank -= numOfAllLowData[0];
                timeI++;     // rank=
        }
        timeI += 2;       // dataIsInLeftPart=, if
}

// 目的のデータがどちらの部分データに存在するかを他のプロセッサに伝える
// このメソッドはプロセッサ 0 のみが実行する
public void broadcastWhereIsTarget() {
        for (int i=1; i<numOfProcessors; i++)
                sendBool(i, dataIsInLeftPart);
        timeI += numOfProcessors;    // for
}

// 目的のデータがどちらの部分データに存在するかを受信する
public void receiveWhereIsTarget() {
        dataIsInLeftPart = receiveBool ();
        clearNetwork ();
}

// 目的のデータが含まれる部分データを選ぶ
// 配列 data を目的のデータが含まれる部分データに縮小する
public void selectSubData() {
        if (dataIsInLeftPart) {   /* 目的のデータが基準値未満の部分に存在するとき */
                if (lowRight >= 0) {
                        int tmp[] = new int[numOfLowData];
                        for (int i=0; i<tmp.length; i++)
                                tmp[i] = data[i];
                        data = tmp;
                        timeI += (tmp.length)*2+2; // tmp[]=, for, tmp[]=, data=
                } else {
                        data = null;
```

```
                timeI++;                        // data=
            }
        } else {                    /* 目的のデータが基準値以上の部分に存在するとき */
            if (highRight >= highLeft) {
                int tmp[] = new int[numOfHighData];
                for (int i=0; i<tmp.length; i++)
                    tmp[i] = data[lowRight+i+1];
                data = tmp;
                timeI += (tmp.length)*2+2; // tmp[]=, for, tmp[]=, data=
            } else {
                data = null;
                timeI++;                        // data=
            }
        }
}

// データをプロセッサ 0 に送信する
public void sendData() {
        if (data != null)
            sendArray (0, data);
        timeI++;    // if
}

// データを受信する
// このメソッドはプロセッサ 0 のみが実行する
public void receiveData() {
        data = receiveArray ();
}

// 逐次選択
// このメソッドはプロセッサ 0 のみが実行する
public void sequentialSelection() {
        target = select(data, 0, data.length-1, rank);
}

// 配列 a の low 番目から high 番目までを基準値 piv 以下の部分と以上の部分に分割し、
// lowLeft, lowRight, highLeft, highright の値を設定する
// 実行後は lowRight+1 == highLeft または lowRight+2 == highLeft, lowLeft == 0, highRight ==
data.length-1 となっており
```

```
// a[lowLeft]～a[lowRight]に piv 以下の値、a[highLeft]～a[highRight]に pivot 以上の値が入っている
// lowRight+2 == highLeft  とのきは、a[lowRight+1] == pivot となっている
void divide(int[]a, int low, int high, int piv) {
        int i=low, j=high;
        timeI+=2;

        while (i<=j) {
                timeI++;                        // while

                while (a[i] < piv) { // piv より大きい数を発見するまで走査
                        i++;
                        if (i>j) break;
                        timeI+=3;               // while, i++, if;
                }
                while (a[j] > piv) { // piv より小さい数を発見するまで走査
                        j--;
                        if (i>j) break;
                        timeI+=3;               // while, j--, if
                }

                if (i<=j) {
                        swap (a, i, j);   // a[i]と a[j]の値を入れ替える
                        i++;
                        j--;
                        timeI+=2;               // i++, j--
                }
                timeI++;                        // if
        }
        /* ループ終了時は  j+1==i  または  j+2==i  となっており  */
        /* a[low]～a[j]に piv 以下の数,a[i]～a[high]に piv 以上の数が入っている  */
        /* j+2==i  のときは、a[j+1] == piv  となっている  */

        lowLeft = low;            // データ基準値以下の部分の左端
        lowRight = j;             // データ基準値以下の部分の右端
        highLeft = i;             // データ基準値以上の部分の左端
        highRight = high;         // データ基準値以上の部分の右端
        timeI+=4;                       // lowLeft=, lowRight=, highLeft=, highRight=
}
```

```
// 配列 a の中央値を選ぶ
int selectMedian (int[] a) {
        return select (a, 0, a.length-1, a.length/2);
}

// 配列 a の low 番目から high 番目までの中で k-low+1 番目に小さいデータを選ぶ
int select (int[] a, int low, int high, int k) {
        return semiSelect (a, low, high, k, 0);
}

// 配列 a の a.length/2±a.length/8 番目に小さいデータを選ぶ
int semiSemiMedian (int[] a) {
        return semiSelect (a, 0, a.length-1, a.length/2, a.length/8);
}

// 配列 a の low 番目から high 番目までの中で k-low+1±r 番目に小さいデータを選ぶ
int semiSelect (int[] a, int low, int high, int k, int r) {
        if (high-low < 0) {                  // 解候補が範囲内に存在しないとき
                timeI++;                              // (アルゴリズムが正しければこれは起こりえない 9
                return Integer.MAX_VALUE;  // デバグ用に無限大を返す
        } else if (high-low == 0) {      // 解候補が 1 個のとき
                timeI++;
                return a[low];
        } else if (high-low <= r) {      // 解候補が r+1 個以下のとき
                int   i = (int) (Math.random()*(high-low+1)); // 候補の中からランダムに選ぶ
                timeI++;
                return a[low + i];
        } else if (high-low == 1) {      // 解候補が 2 個のとき (r<1 のときこのケースが起こり得る)
                if (a[low] > a[high])
                        swap (a, low, high);     // a[low]と a[high]の値を入れ替える
                timeI++;
                if (low == k) return a[low]; else return a[high];
        } else {                                  // 解候補が r+2 個以上(r<1 のときは 3 個以上)あるとき
                int piv = a[(low+high)/2]; // 基準値選択
                timeI++;                              // piv=

                divide (a, low, high, piv);  // データを分割する

                int ll = lowLeft;     // データ基準値以下の部分の左端
```

```
                int lr = lowRight;   // データ基準値以下の部分の右端
                int hl = highLeft;   // データ基準値以上の部分の左端
                int hr = highRight; // データ基準値以上の部分の右端
                timeI+=4;                       // ll=, lr=, hl=, hr=
                // lowLeft, lowRight, highLeft, highRight の値は select を再帰呼び出しする度に書き換えられるので
                // 作業用変数にそれぞれの値のコピーを取って置く

                timeI+=2;                       // if, if
                if (lr>=k) {                                    // k-low+1 番目に小さい数が前半にあるとき
                        piv = select (a, ll, lr, k);
                        timeI++;                // piv=
                } else if (hl<=k) {                             // k-low+1 番目に小さい数が後半にあるとき
                        piv = select (a, hl, hr, k);
                        timeI++;                // piv=
                } else {
                        piv  =  a[k];                                   // 基準値が k-low+1 番目に小さい数だったとき
                        timeI++;                        // piv =
                }
        return piv;
        }
}

// 配列 a から m 個のサンプルを取り、その中央値を返す
int sampleMedian (int [] a, int m) {
        if (m <= 0) {
                timeI++;                        // if
                return Integer.MAX_VALUE;       // m<=0 の場合はデバグ用に無限大を返す
        } else if (m <= 2) {
                timeI++;                        // if
                return a[(int) (Math.random() * a.length)]; // m<=2  の場合はランダムな要素を返す
        } else if (m == 3) {
                timeI++;                        // if
                int i = (int) (Math.random() * a.length);
                int j = (int) (Math.random() * a.length);
                int k = (int) (Math.random() * a.length);
                timeI+=3;                       // i=, j=, k=
```

```
                if (a[i] <= a[j]) {
                        if (a[k] <= a[i]) {
                                timeI+=2;          // if, if
                                return a[i];
                        } else if (a[j] <= a[k]) {
                                timeI+=3;          // if, if, elseif
                                return a[j];
                        } else {
                                timeI+=3;          // if, if, elseif
                                return a[k];
                        }
                } else {
                        if (a[k] <= a[j]) {
                                timeI+=2;          // if, if
                                return a[j];
                        } else if (a[i] <= a[k]) {
                                timeI+=3;          // if, if, elseif
                                return a[i];
                        } else {
                                timeI+=3;          // if, if, elseif
                                return a[k];
                        }
                }
        } else {
                int b[] = new int[m];
                for (int i=0; i<m; i++)
                        b[i] = a[(int) (Math.random() * a.length)];
                timeI+=m;

                return selectMedian (b);
        }
}

// a[i]と a[j]の値を入れ替える
void swap (int[] a, int i, int j) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
        timeI+=3;
```

```
}

// プロセッサ p に int 型データ n を送信する
void send (int p, int n) {
        network.put (p, n);
        timeG++;
}

// プロセッサ p に int 型配列データ a を送信する
void sendArray (int p, int[] a) {
        network.putArray (p, a);
        timeG += a.length;
}

// プロセッサ p に boolean 型データ b を送信する
void sendBool (int p, boolean b) {
        network.putBool (p, b);
        timeG++;
}

// int 型データを受信する
int receive () {
        int n;
        if (network.isEmpty(processorNumber)) n = Integer.MAX_VALUE;
        else    n = network.get (processorNumber);
        timeI++;
        timeG++;
        return n;
}

// int 型配列データを受信する
int[] receiveArray () {
        if (network.isEmpty (processorNumber)) {
                timeI++;
                timeG++;
                return null;
        } else {
                int[] a = network.getArray (processorNumber);
                timeI++;
```

```
                timeG += a.length;
                return a;
        }
}

// boolean 型データを受信する
boolean receiveBool () {
        boolean b = network.getBool (processorNumber);
        timeG ++;
        return b;
}

// 受信キューをクリアする
void clearNetwork () {
        network.clear (processorNumber);
}

// 保持するデータ表示
public void showData() {
        if (processorNumber < 10)
                System.out.print ("Pr. "+processorNumber+": ");
        else System.out.print ("Pr."+processorNumber+": ");
        System.out.println (formatDataList (data));
}

// 保持するデータ出力
public void printData() {
        if (processorNumber < 10)
                output.print ("Pr. "+processorNumber+": ");
        else output.print ("Pr."+processorNumber+": ");
        output.println (formatDataList (data));
}

// 保持するデータ出力(ログ出力用)
public void logData() {
        if (processorNumber < 10)
                log.print ("Pr. "+processorNumber+": ");
        else log.print ("Pr."+processorNumber+": ");
        log.println (formatDataList (data));
```

```
}
// 目的データ表示
// このメソッドはプロセッサ 0 のみが実行する
public void showTarget() {
        System.out.println ("Target : " + target);
}

// 目的データ出力
// このメソッドはプロセッサ 0 のみが実行する
public void printTarget() {
        output.println ("Target : " + target);
}

// 目的データ出力(ログ出力用)
// このメソッドはプロセッサ 0 のみが実行する
public void logTarget() {
        log.println ("Target : " + target);
}

// 目的データ数表示
// このメソッドは通常使われない(デバグ用)
public void showNumOfData() {
        if (processorNumber < 10)
                System.out.print("Pr. "+processorNumber+": ");
        else System.out.print("Pr."+processorNumber+": ");
        if (data == null) System.out.println (0);
        else System.out.println (data.length);
}

// 目的データ数出力
// このメソッドは通常使われない(デバグ用)
public void printNumOfData() {
        if (processorNumber < 10)
                output.print("Pr. "+processorNumber+": ");
        else output.print("Pr."+processorNumber+": ");
        if (data == null) output.println (0);
        else output.println (data.length);
}
```

```
// 目的データ数表示(ログ出力用)
// このメソッドは通常使われない(デバグ用)
public void logNumOfData() {
	if (processorNumber < 10)
		log.print("Pr. "+processorNumber+": ");
	else log.print("Pr."+processorNumber+": ");
	if (data == null) log.println (0);
	else log.println (data.length);
}

// ランク表示
// このメソッドはプロセッサ 0 のみが実行する
public void showRank() {
	System.out.println ("Rank : " + rank);
}

// ランク出力
// このメソッドはプロセッサ 0 のみが実行する
public void printRank() {
	output.println ("Rank : " + rank);
}
// ランク出力(ログ出力用)
// このメソッドはプロセッサ 0 のみが実行する
public void logRank() {
	log.println ("Rank : " + rank);
}
// 出力用にデータ配列を整形
String formatDataList (int[] d) {
	if (d == null)
		return "Empty";
	else {
		String str = "";
		for (int i=0; i<d.length; i++)
			str += formatData (d[i]);
		return str;
	}
}
// 出力用にデータを整形
String formatData (int d) {
```

```
            if (d < 10) return "    " + d;
            else return " " + d;
    }
}
/* BspProcessor.java ここまで  */
```