卒業研究報告書

題 目

BSP モデルを用いた並列計算の有用性の検証

指導教員

石水隆助手

報告者

00-1-26-019

伊波 修一

近畿大学理工学部電気工学科

平成１７年２月 19 日提出

目次

第1章　序論

1.1.　並列計算機の概要

現在、地震データの解析や天体現象のシミュレーションなどコンピュータ処理の対象となるデータの情報量が膨大である問題が存在する。これらの問題に対して従来のようなプロセッサが１つである逐次型コンピュータを用いて計算するのは膨大な時間がかかり有効とは言えない。特に地震情報などは早期にデータを処理する必要がある。
このような問題に対して有効な手段として考えられるのが並列計算機である。並列計算機とは従来の逐次型計算機とは異なり、ネットワークなどを介して複数のプロセッサが互いに情報をやりとりしながら並列に問題を処理するものである。複数のプロセッサを用いるため、より複雑な問題に対しても高速に解くことが出来る。加えて近年ではプロセッサを安価に手に入れることが出来るために並列計算機を構築することが比較的容易となってきている。

並列計算機は、共有メモリ型計算機（Shared Memory Parallel Computer）と分散メモリ型並列計算機(Distributed Memory Parallel Computer)に分けられる。共有メモリ型並列計算機は共有メモリ(Shared Memory)とそれに接続された複数のプロセッサから成る。一方、分散メモリ型並列計算機は、局所メモリ(Local Memory)を持つ複数のプロセッサとそれらを結びつけるネットワークから成る。

一般的に共有メモリ型並列計算機は通信遅延が短くプロセッサ間の同期も取りやすいが、プロセッサを増やすことは困難である。逆に分散メモリ型並列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は大きな時間がかかる。このためプロセッサ数が少ない計算機は共有メモリ型か、プロセッサ数が多い並列計算機は分散メモリ型が主流となっている。

1.2.　並列アルゴリズムと並列計算モデル

並列計算機は対象となる問題をよりサイズの小さい部分問題に分割し、各プロセッサがそれぞれの部分問題を同時に処理することにより高速化を行える。しかし並列計算機はデータ通信やプロセッサの同期など逐次型計算機には無い概念があり、逐次型計算機のアルゴリズムをそのまま並列計算機に載せかえることは出来ず、並列計算機用のアルゴリズムを新たに設計する必要がある。

並列計算機用のアルゴリズムである並列アルゴリズムは、並列計算機を抽象化した並列計算モデル(Parallel Computing Model)上で設計、解析される。並列計算モデルはメモリの形式、ネットワークの形状、通信遅延の特性などが異なる様々なモデルである。本研究では代表的な並列計算モデルである PRAM(Parallel Random Access Machine)

および近年注目されている並列計算モデルである BSP(Bulk Synchronous Parallel)モデルを扱う。

1.3. 本研究の目的

本研究では BSP モデル上で高速な整列アルゴリズムの提案を行う。整列は様々な分野で利用される重要な基本問題の１つであり、BSP モデル上においても高速な整列アルゴリズムが必要とされている。

第2章　原理

2.1.　PRAM (parallel random access machine)モデル

PRAM (parallel random access machine)モデルは複数のプロセッサがメモリを共有するモデルである。図 1 に PRAM 組織図を示す。なお P はプロセッサを意味する。PRAM の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内のデータを 1 単位時間で読み書きできる。また、全ての計算は 1 単位時間で行うことが出来る。加えて PRAM は細粒度同期式であり、一単位時間ごとに全てのプロセッサで同期が取られる。

このように並列処理機構が理想的に設定されているため，　並列アルゴリズムの設計を容易なものにし、問題の並列性をある程度理論的に検証することを可能にしている。更に PRAM は他の並列計算機モデルの基礎となることも多く PRAM を対象に設計されたアルゴリズムは数多く存在する。しかし、PRAM は通信コストなどを考慮しない理想的なモデルであるため現実とのギャップがあり、これらのアルゴリズムを実行できる効率の良い並列計算機は存在しないのが現状である。

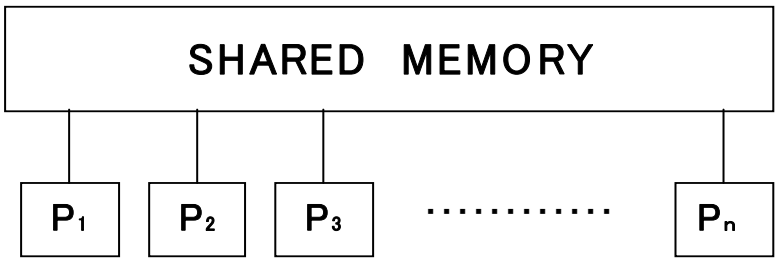

図１　並列ランダムアクセス機械

Fig.1 Parallel Random Access Machine

2.2.　BSP (bulk synchronous parallel)モデル

BSP (bulk synchronous parallel)とは通信コスト、同期間隔等を考慮した非同期式分散メモリ型並列計算モデルである。BSP モデルは局所メモリを持つ複数のプロセッサとそれらを結びつけるネットワークおよびプロセッサ間でバリア同期を取るための同期機構からなる。プロセッサ間の通信はネットワークを通してメッセージ交換をすることにより行われる。第 2 図に BSP の組織図を示す。

従来の PRAM は通信コストや同期を考えない理想的なモデルであった。初期の並列計算機ではプロセッサの演算能力が低く、プロセッサの内部演算時間に比べてプロセッサ間の通信はさほど考慮させていなかった。しかしプロセッサ演算能力の向上に伴い、通信コストが並列計算における処理時間の重大な要素となっている。この為、共有メモリ型並列計算機モデルは近年の並列計算機とは現実との大きなギャップがある。BSP はこのギャップを埋めるべく提案されたものである。BSP モデルは通信遅延や同期時間、通信帯域幅等を表すために以下に示すパラメタをもつ。

- P　　：プロセッサ数。各プロセッサには１～P の識別番号が割り当てられ P_1、P_2、・・・・・P_p と表す。
- g　　：１個のメッセージを送信するのにかかる時間
- L　　：通信遅延時間。これはまた、バリア同期にかかる時間も表す。

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成る。各スーパーステップは内部計算命令の列から成る内部計算フェーズで、送信あるいは受信命令の列から成る通信フェーズで構成されており、各プロセッサは割り当てられたスーパーステップを非同期に実行する。スーパーステップの命令の終了後、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。あるスーパーステップで各プロセッサが各々w 個の内部計算命令と各々 h 個の通信命令を実行する場合、そのスーパーステップの時間計算量は O(w+gh+L)であると仮定されている。

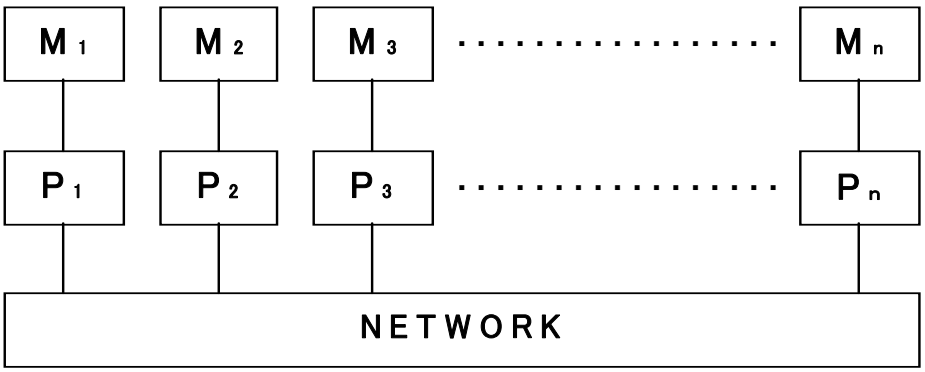

図２　ＢＳＰ組 織 図

Fig2. BSP (Bulk Synchronous Parallel) model

2.3.　クイックソート

本研究で提案するアルゴリズムはクイックソートをベースとして BSP モデル上の実行をシミュレートするものである。クイックソートとは現在最もよく使われている整列アルゴリズムの一つであり、最速であるとされている。一般的な手法は与えられた配列

の中から任意の要素を一つ取り出して、その値を基準とし配列をその要素より値が小さい部分配列とそれ以外の要素と部分配列の二つに分ける。次に、前の手順で作った二つの部分配列に対してそれぞれ同様の操作を行い、これを繰り返して整列させるというものである。入力配列 A を部分配列 A_1、A_2 に分割したとき A_1、A_2 にそれぞれ 1 台のプロセッサを割り当てれば並列処理を行うことが出来る。以下同様に、配列が分割されるたびに各部分配列に 1 台のプロセッサを割り当てるという操作を部分配列の個数が p 個になるまで繰り返す。部分配列の個数が p 個になれば各部分配列に 1 台のプロセッサが割り当て逐次にソートを行えばよい。PRAM では配列 A を 2 個の部分配列 A_1、A_2 に分割する場合、A_1 の処理時間と A_2 の処理時間が等しいときが最も効率的になる。つまり A を分割する基準値として A 内のデータの中央値を用いることが最も好ましい。

しかし基準値がソート対象となる配列の中央値から大きく外れたものであれば極端に分割され効率が悪くなる。極端な分割のされ方がなされるとプロセッサに依ってその計算量のアンバランスが生じる。このことは同期が必要である並列計算では計算量が増大する大きな要因となる。

中央値を求めるには選択問題を解けば良い。しかし、クイックソートの中央値は厳密に中央である必要はなく、中央値に近い値であれば十分である。そこで配列から少数のサンプルを選び、そのサンプルの中央値を基準値として用いる。サンプルの数(S)を増やすと基準値の精度が上がるが基準値を決定する為に必要となる時間が増大するので大きければ良いというわけではなく、ソーティングの対象となるデータによって最適な S が変わる。

一方、BSP モデル上では通信時間を考慮しなければならないため、中央値を基準値とした場合、効率よく実行できるとは限らない。プロセッサ P_1 が配列 A を部分配列 A_1、A_2 に分割した後、プロセッサ P_1 が引き続き A_1 を処理し、プロセッサ P_2 が A_2 を処理するとする。このとき A_2 を P_1 から P_2 に送信しなければならない。P_1、P_2 により A_1、A_2 の処理時間をそれぞれ t_1、t_2、P_1 から P_2 へ A_2 を送信するのに要する時間を t_3 とすると、$t_1 = t_2 + t_3$ である。

第3章　方法

3.1.　クイックソートシミュレートプログラム

本研究では、BSP モデル上で高速に並列クイックソートを行うため、基準値の選択法を提案する。付録１に本研究で作成したプログラムを示す。このプログラムはデータ数 n、プロセッサ数 p、通信帯域幅 g、通信遅延時間 L が与えられたとき BSP モデル上でクイックソートを実行させることができ、指定した位置の基準値を用いてデータ分割したときの実行時間を計測するプルグラムである。

本研究で作成したクイックソートシミュレートプログラムは、inputArray, outputArray, quickSort, select, という４つの関数を用いる。これらの内容を以下に示す。

inputArray()： 任意の範囲内の乱数を任意の数だけ作成する。

outputArray()： 作成した乱数と整列させた数を画面上に表示する。

select()： 配列から任意の数だけサンプルを取りそれらを整列させ、その中から X 番目に小さい値を探す。なお X には任意の値を入力できる。

quickSort()： select 関数を用いて X 番目に小さい値を基準値とし、配列を整列させる。この作業をプロセッサが複数ある場合は通信コスト等を考慮した並列計算をシミュレートする。

3.2.　実験方法

付録に示したプログラムをファイルを用いて、プロセッサ台数 p、データ数 n, サンプル数 S, 基準値の取り方 X(サンプル数 S 個の小さい方から X 番目の値を基準値とする)、同期周期 L、通信命令実行時間 g を変えながら実行し BSP モデル上での実行時間を計測する。

第4章　結果

4.1.　最適なサンプル数

ソーティングの対象としたデータによって最適なサンプル数(S)が変化する。この実験では、0～99 の範囲でランダムな数を 1000 個生成したものをソーティングの対象とした。整列させる為に 2 つのデータを 1 回比較するのに要する内部計算量、およびサンプルを 1 個取り出すのに要する内部計算量を 2、L,g をそれぞれ 16 と 32、X=S/2 に設定し、S と P を変更し実験を行った結果を表 1 および図 3 に示す。

表 1 BSP モデルシミュレートの実験結果（$L=16, g=32, X=\frac{S}{2}$）

	P=1	P=2	P=4	P=8	P=16	P=32	P=64	P=128	P=256	P=512
S=2	54,910	52,984	52,601	52,413	58,808	56,597	58,487	57,196	58,925	57,996
S=4	53,969	53,221	57,412	56,881	58,224	54,571	53,109	52,305	53,224	57,301
S=8	55,469	56,416	53,677	47,803	47,803	46,820	50,248	52,684	48,851	48,352
S=16	55,352	50,329	43,546	42,523	43,048	44,557	45,427	47,341	47,286	43,907
S=32	55,219	49,613	42,519	41,465	42,580	41,584	42,846	42,564	41,252	42,586
S=64	55,382	47,905	44,729	44,513	43,483	43,539	42,859	43,296	43,584	43,895

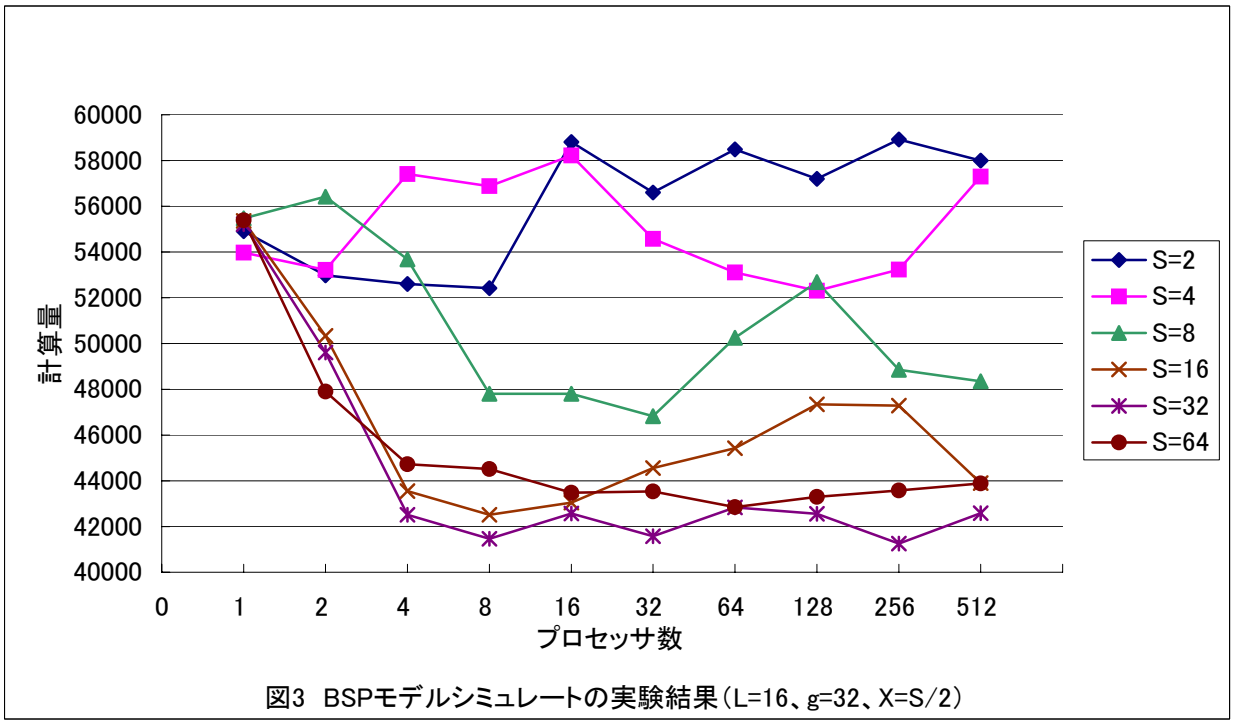

図3　BSPモデルシミュレートの実験結果（L=16、g=32、X=S/2）

この結果からサンプル数＝32 が最適であると判明した。

4.2. 通信帯域幅が狭いときのシミュレーション

通信帯域幅が狭いときのシミュレーションとして L=16、g=32、S=32 と設定しプロセッサ P と、基準値の取り方を決める X の値を変更し実験を行った。実験結果は 4.1 と同様に 0～99 の範囲でランダムな数を 1000 個生成したものをソーティングの対象とした。この結果を表 2 および、図 4 に示す。

表 2 BSP モデルシミュレートの実験結果（L=16、g=32、S=32）

	1	2	4	8	16	32	64	128	256	512
X=16	55,219	49,613	42,519	41,465	42,580	41,584	42,846	42,564	41,252	42,586
X=14	56,413	42,799	36,235	34,452	33,582	31,104	34,788	34,272	35,012	34,815
X=12	56,356	41,582	35,127	29,697	28,750	27,581	28,022	29,401	30,248	28,792
X=10	57,109	43,120	33,985	26,616	26,478	25,646	24,776	24,236	24,831	26,266
X=8	57,243	42,400	34,396	27,529	25,549	23,878	21,516	20,816	22,789	22,958
X=6	56,084	49,305	39,851	35,181	32,549	26,512	23,839	22,798	22,586	23,905

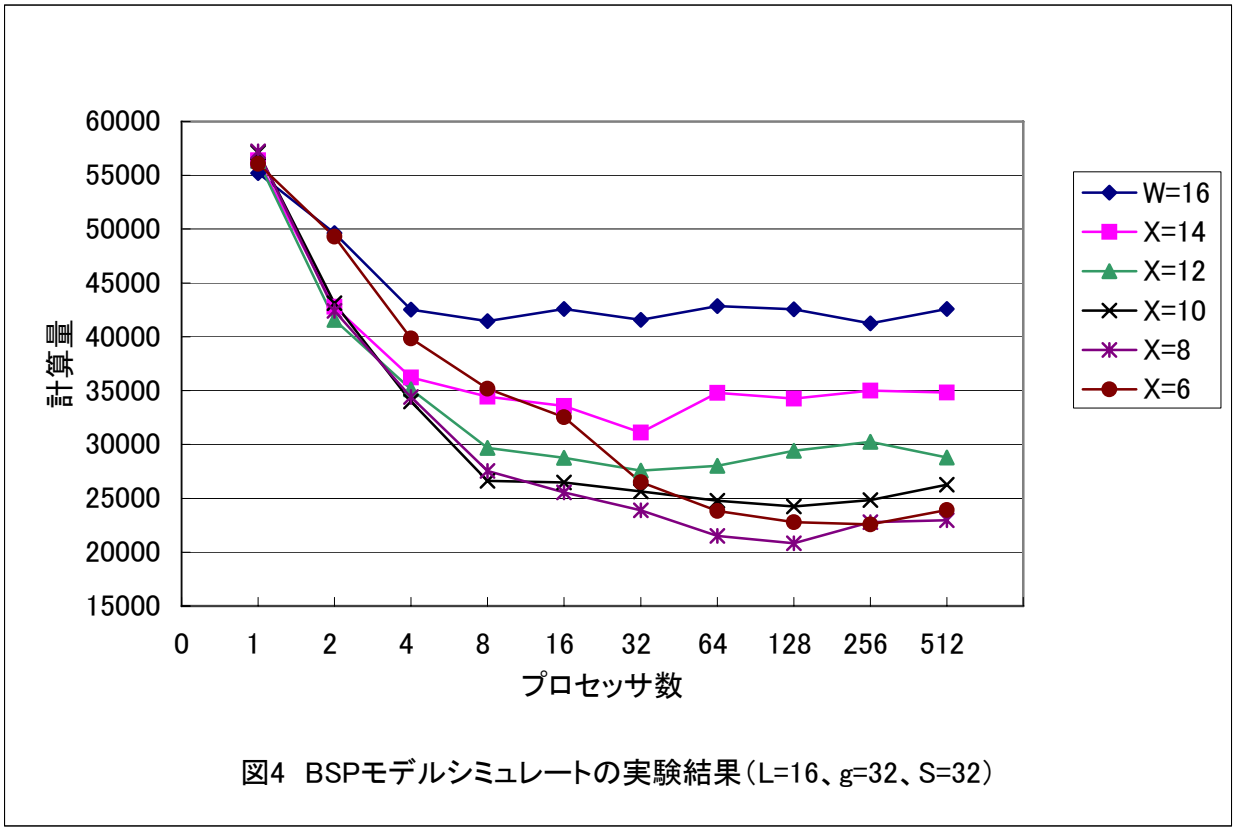

図4 BSPモデルシミュレートの実験結果（L=16、g=32、S=32）

4.3. 通信帯域幅が広いときのシミュレーション

通信帯域幅が広いときのシミュレーションとして L=16、g=8、S=32 と設定しプロセッサ P と、基準値の取り方を決める X の値を変更し実験を行った。実験は 4.1、4.2 と同様に 0～99 の範囲でランダムな数を 1000 個生成したものをソーティングの対象とした。この結果を表 3 および、図 5 に示す。

表 3 BSP モデルシミュレートの実験結果（L=16、g=8）

	P=1	P=2	P=4	P=8	P=16	P=32	P=64	P=128	P=256	P=512
X=16	55,219	35,514	27,486	22,108	20,149	18,168	17,368	17,214	17,299	17,375
X=14	56,413	35,370	24,806	19,147	16,746	16,352	15,936	15,751	14,915	14,920
X=12	56,356	36,525	26,686	19,645	16,912	14,920	14,825	14,150	14,852	14,997
X=10	57,109	35,101	28,697	21,840	17,358	15,765	14,905	14,599	14,372	15,470
X=8	57,243	42,096	36,009	29,418	24,389	19,754	16,754	16,690	16,464	15,510

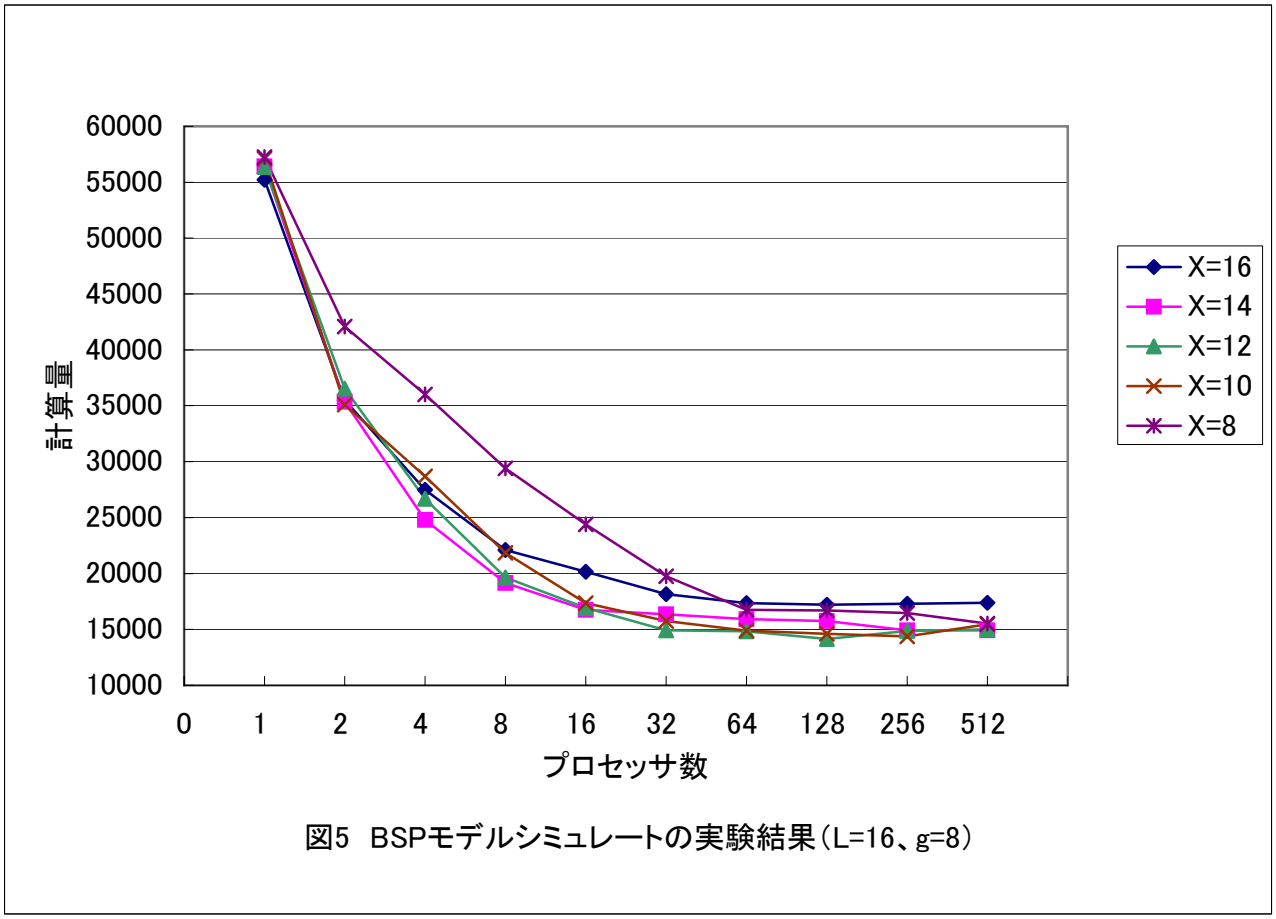

図5 BSPモデルシミュレートの実験結果（L=16、g=8）

第5章　考察

5.1.　計算量の考察

表 1、図 3 から S=2,4 と S=32,64 を比較すると前者はプロセッサ数が大きくなっても計算量が徐々に小さくなることはなく計算量に大きなバラツキがある。このことはサンプル数が少ない場合、基準値が配列の中央値の周囲にひどく不規則に分布したため、分割のされ方が歪になり効率が悪くなったためであると考えられる。

また表 2、図 4 からプロセッサ台数が少ないうちは基準値の取り方による差が比較的少ない。これはプロセッサ台数が少ない場合、通信する回数がすくないために分割の仕方にあまり影響を受けなかったため、計算量の差が少なくなったと考えられる。また S の中央値である X=16 の場合、P=8 から先はプロセッサ数が増えてもその計算量はほぼ横ばいである。これは通信帯域幅が狭いため、効率の悪い基準値を選んだ場合、通信時間が多くかかってしまいプロセッサ数を増やしても計算量の減少につながらなかったと考えられる。そして最も計算量が少なかったのが X=8 のときである。これはサンプル数 32 個の前から 8 番目の値を基準値とした場合であり、その分割の割合は 1:3 である。

つまり g=32、L=16 のとき BSP モデル上の並列クイックソートでデータを分割する際は 1:3 の割合で分割するのが最適となる。

次に通信帯域幅が狭いときのシミュレーションである図 4 と通信帯域幅が広いときのシミュレーションである図 5 を比較検討する。通信帯域幅が狭い前者に比べ通信帯域幅が広い後者は基準値の取り方にさほど影響を受けずに P=32 付近まではプロセッサ数の増加に伴い計算量が減少している。また、後者で最も計算量が少なかったのは X=12 の時であり、S=32 であるから分割の割合は 3:5 である。

つまり g=16、L=8 のとき BSP モデル上の並列クイックソートでデータを分割する際は 3:5 の割合で分割するのが最適となる。

第6章　結論

6.1.　結論

BSP モデル上で並列クイックソートを行う場合、通信帯域幅によって配列を分割する割合である最適な X の値が変化する。通信帯域幅の広い場合は通信にかかる時間が少ないため計算量はこの X の値に比較的左右されにくい。しかし通信帯域幅が狭い場合は通信により多くの時間を要するために計算量は X の値によって大きく変化する。このため通信帯域幅が狭い場合は配列を分割する割合が計算量に対して特に重要な要素である。

参考文献

1) L.G. Valiant: “A Bridging Model for Parallel Computing,” Comm. Of the ACM(1990)
2) J. JáJá: “An Introduction to Parallel Algorithms,” Addison Wesley Publishing Company (1992)

謝辞

本研究の報告書を作成するにあたり、石水隆先生には多大な御指導、御支援を頂き、大変感謝しております。また、制御回路工学研究室の方に配慮を頂き、感謝の意を表します。誠に有難うございます。

付録

```
/* BSP モデル上におけるソーティングのシミュレーション */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#define N 1000          /* データ数の上限 */
#define M 100           /* 値の範囲 */
#define P 512           /* プロセッサ数の上限 */
#define S 32            /* 分離子選択のサンプルの個数 */
#define X 12            /*  X 番目を基準値 */
#define G 8             /* 通信命令実行時間 g  */
#define Z 16            /* 同期周期 L  */


main() {
        int a[N];
        int i, t, t2;
        void inputArray(), outputArray();
        int quickSort();

        t2=0;

        for(i=0; i<=10000; i++){

        inputArray(a,N);

        t = quickSort(a, 0, N-1, 0, P);

        t2 += t;
        }

        printf("Time : %d¥n", t2/10001);
}
```

```
void inputArray(a,n) int *a,n; {                    /*   入力データ作成   */
        int i;

        srand((unsigned int)time(NULL));            /* 乱数の初期化 */
        for(i=0; i<n; i++) a[i] = rand()%M;          /* 0～m-1 の乱数を発生 */
}

void outputArray(a,n) int *a,n;{                    /*   データ出力   */
        int i;

        printf("Data : ");
        for(i=0; i<n; i++) printf("%d ", a[i]);
        printf("¥n");
}



int quickSort(a, l, r, p, pp) int *a, l, r, p, pp; {        /*quicksort 関数の宣言*/
        int i, j, t, t1, t2, g, L;
        int pivot;
        int b[S];
        int swap(), select();

        t=0; t1=0; t2=0;
        g=G;
        L=Z;


        if((r-l)>S) {
                for(i=0; i<S; i++)                  /* S 個の配列 b を作る */
                b[i] = a[l+i];                      /* a[l]～a[r]からサンプルを S 個抽出 */
                t=t+2;
        t=t+select(b,0,S-1,X,&pivot);
                        /* 配列 b の b[0]～b[S-1]の中で X 番目に大きいデータを探す*/
        }
        else {
                pivot = a[(l+r)/2];                 /* 適当な値を基準値にする */
                t++; /* pivot= */
        }
```

```
t++; /* if */


i=l;
j=r;
t=t+2; /* m=, pivot=, i=, j= */

do {
        t++; /* do */
        while(a[i] < pivot){
                i++;
                t=t+2; /* while, i++ */
        }
        t++; /* while */
        while(a[j] > pivot) {
                j--;
                t=t+2; /* while,i-- */
        }
        t++; /* while */
        if(i <= j) {
                t = t+swap(a,i,j);
                i++;
                j--;
                t=t+2; /* i++,j++ */
        }
        t++; /* i++, j++ */
}

while(i <= j);
if(pp >= 2) { /* 複数のプロセッサで分担 */
        if(l<j) t1=quickSort(a, l, j, p, pp/2)+((j-l)*g)+L;
                        /* 再帰：　プロセッサ p による a[l]～a[j]間のソート */
        if(i<r) t2=quickSort(a, i, r, (p+pp)/2, pp/2);
                        /* 再帰：　プロセッサ p+pp/2 による a[i]～a[r]間のソート */
        t = t+2; /* if,if */
        if(t1 > t2) t=t+t1; else t=t+t2;
                /* プロセッサ p と p+pp/2 のうち実行命令数の多いほうを足す */
}
else {          /* 1 台のプロセッサですべて実行 */
```

```
		if (l<j) t=t+quickSort(a, l, j, p, 1);
					/* 再帰：　プロセッサ p による a[l]～a[j]間のソート */
		if (i<r) t=t+quickSort(a, i, r, p, 1);
					/* 再帰：　プロセッサ p による a[i]～a[r]間のソート */
		t = t+2; /* if,if */
	}
	t++; /* if */

	return t;
}
					/*quicksort 関数の宣言の終了*/


					/*select 関数の宣言の終了*/
int select(a,l,r,m,o) int *a,l,r,m,*o; {
				/* a[l]～a[r] の中で m-l+1 番目に大きいデータを探す */
				/* 解は o に入れられる */
	int i,j,t=0;
	int pivot;
	int swap();

	pivot = a[(l+r)/2]; /* 適当な値を基準値にする */
	i=l;
	j=r;
	t=t+2; /* pivot=, i=, j= */

	do {
		t++; /* do */
		while(a[i]<pivot) {
			i++;
			t=t+2; /* while,i++ */
		}
		t++; /* while */
		while(a[j]>pivot) {
			j--;
			t=t+2; /* while,i-- */
		}
		t++; /* while */
		if(i<=j) {
```

```
                t=t+swap(a,i,j);
                i++;
                j--;
                t=t+2; /* i++,j++ */
            }
            t++; /* if */
        } while(i<=j);
        if(l<j && l<=m && m<=j) t=t+select(a,l,j,m,o);
                        /*  再帰: a[l]～a[j]間で m-l+1 番目に小さいデータを探す  */
        if(i<r && i<=m && m<=r) t=t+select(a,i,r,m,o);
                        /*  再帰: a[i]～a[r]間で m-i+1 番目に小さいデータを探す  */
        t=t+2; /* if,if */

        *o = a[m]; /*  解の出力  */
        t++; /* *o= */

        return t;
}
                                /*select 関数の宣言の終了*/


int swap(a, i, j) int *a, i, j;{            /*   データの入れ替え   */
        int tmp;

        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;

        return 3;
}
```