並列計算機とプログラミング

秋田県立大学 電子情報システム学科

小澤 一文　廣田 千明　中村 真輔

目 次

1 はじめに

1.1 並列処理

- 並列計算の目的は
  一つの問題を解くために必要な時間を複数個のプロセッサ (CPU) を持つ並列計算機を用いて減らすこと。
- 並列計算機とは
  マルチコンピュータ (多くのコンピュータがネットワークで繋がれている) と，多くの CPU が一つの筐体の中にあり，メモリを共有している **SMP** (symmetric multiprocessor) に分けられる。
- 並列プログラミング
  計算の異なる部分を異なるプロセッサに割り当てて，それぞれを同時に計算できるよう，明示的に書ける言語 (代表的なものに OpenMP と MPI がある) を用いたプログラミングを言う。

1.2 科学的手法の変化

古典的な科学は，観測，理論，実験，からなっていた。これは実験とモデルによって特徴づけられる。しかし実験するには時間がかかりすぎるか，コストがかかりすぎる場合が多くある。これに対して計算機の発明後の，現代の科学は，観測，理論，実験，**数値シミュレーション** からなっている。しかし多くの科学上の問題は非常に複雑で，数値シミュレーションを通して解決するにはとてつもなく強力な計算機が必要になる。

以下の分野ではスーパーコンピュータの活躍が期待されている (grand challenge)：

1. 量子化学，統計力学，相対論
2. 宇宙論，天文学
3. 流体力学
4. 物質設計，超伝導
5. 生物学，薬理学，ゲノム解析
6. 医学，人の細胞組織，骨のモデル化
7. 気象予測，環境問題

2 スーパーコンピュータの歴史

- ENIAC (Electronic Numerical Integrator and Calculator) 1945
  真空管式の最初の汎用計算機，弾道計算に使われた。性能は 350 FLOPS (Floating point operation per second) 程度。

- 最初のスーパーコンピュータ　CRAY-1 (1976)
  パイプライン処理を取り入れた最初の計算機，100 MFLOPS の性能，一台 1000 万$以上，米国の政府機関のみが所有。

- 1970 年代の後半は，民間企業でもスーパーコンピュータが用いられるようになった。

- 1980 年代後半からは世界中の数多くの企業がスーパーコンピュータを用いるようになった。日本製スーパーコンピュータ (富士通 VP シリーズ，日本電気 SX シリーズ，日立 S シリーズ) の活躍があった。

- 50 年間で単体では 10^6 倍高速になった。現在の超並列計算機の性能は 1 TFLOPS (10^{12} FLOPS) 以上である。おおよそ 10^9 倍高速化されている。それはクロックの高速化によるところも大きいが，もう一つは並列性の利用がある。

- スーパーコンピュータの意味が，ベクトル処理 (パイプライン処理) を行う高速計算機から超並列計算機へと変わってきた。

2.1 現代の並列計算機

- 1970 年代後半 VLSI (Very Large Scale Integration) の発明

- Cray-1 は一般企業には高価過ぎるた。実験用の並列計算機はこれに比べればかなり安いが，まだ高価でメンテナンスが個人ではできない。

- 最初の商用並列計算機：Cosmic Cube (1983)
  (Intel 8086+8087) を 64 台結合し，5〜10 MFLOPS の性能を発揮。当時の最高のミニコン VAX 11/780 の 5〜10 の性能だが，価格は半分。

- 商用の並列計算機 (1980 年代後半から 1990 年代)
  多くのベンチャー企業が，並列計算機市場に参入した。例えば，

  > Sequent, nCUBE, Alliant, Silicon Graphics, Kendall Square Research, Sun Microsystems, CRAY, IBM, Encore, Meiko

  など。市場が研究所から一般ユーザへまで拡大していった。

- 各 Vendor は並列計算用のコンパイラ言語も開発したが，ハードウェアの進化に対応していくのが難しいので，C, Fortran などの既存の言語を若干修正し，それに汎用並列ライブラリ (MPI, PVM など) を付け加えるだけという対応になりつつある。

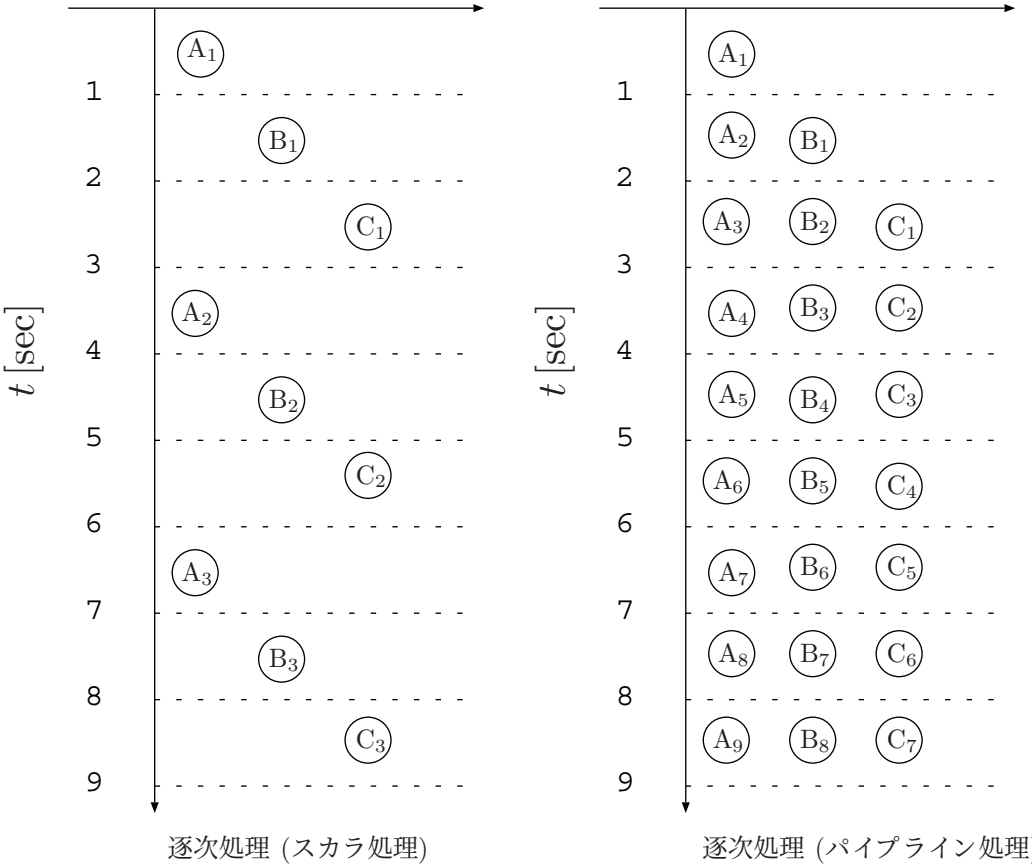
- 市販の部品と Freeware を用いて組み立てた並列計算機が普及している (Beowulf 型並列計算機)。この並列計算機は，商用の並列計算機に比べると安価だが，通信が遅い。

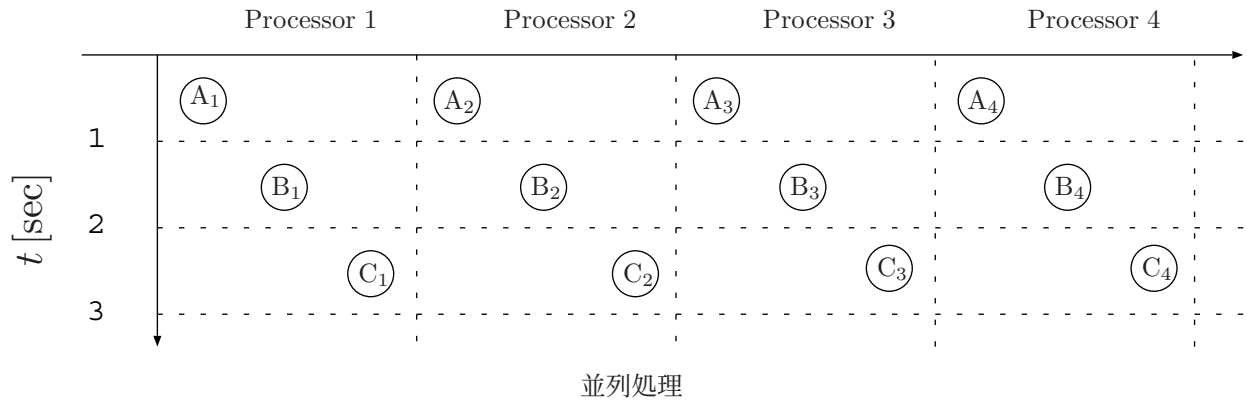
2006 年 11 月におけるスーパーコンピュータの性能比較 (TOP500?? より)

順位	機種名	Vendor	プロセッサ数	ピーク性能 (GFLOPS)
1	BlueGene/L (US)	IBM	131,072	367,000
2	Red Storm (US)	Cray	26,544	127,411
9	TUBAME Grid Cluster (日本)	NEC/SUN	11,088	82,114
14	The Earth Simulator (日本)	NEC	5,120	40,960
500	Blade Cluster (US)	HP (Xeon 3.06GHz)	800	4,896

2.2 パイプライン処理と並列処理

逐次処理の場合でもデータが複数なら高速化できる。例えば，A, B, C という三つタスクを逐次的に行う場合，データがいくつもあるればパイプライン処理によって高速化できる。





データが n 個ある場合，上に示した各処理の時間を比較すると下の表のようになる。ただし，並列処理は n 台のプロセッサで行うものとする。

処理	時間
逐次処理	$3\,n$
Pipeline 処理	$n+2+\alpha$
並列処理	$3+\beta$

上の表で，α, β は，それぞれ パイプライン処理および並列処理に伴って発生する overhead である。

浮動小数点演算をパイプライン処理するハードウェアとそのための命令を持つ計算機を **ベクトル演算** と呼んでいる。ベクトル演算を行うプロセッサを並列につないだ **ベクトル並列計算機** もある。

3 並列計算機のアークテクチャ

3.1 並列計算機の３つのタイプ

- Processor array
  ベクトルプロセッサのように CPU は一つで，計算処理をする部分 (ALU) が多くある。データ並列向きの計算機。プロセッサアレイは同時に同じ演算しかできないので，`if-then-else` などの条件分岐に入ると能率が下がる。

- Multiprocessor
  多くの CPU を持ち，メモリは共有。**UMA** (uniform memory access) processor (SMP とも言う) とも呼ばれている。アドレスが一元的に管理されているので，プログラミングは簡単だが，cache coehrence 問題が生じる。メモリは物理的には分散しているが，論理的には共有という計算機もある。このタイプの計算機は **NUMA** (nonuniform memory access multiprocessor) と呼ばれている。

- Multicomputer
  単体の計算機をネットワークでつないだ計算機。メモリアドレスは一元的でない (同じアドレスが別な場所を示している)。商用の multiprocessor は専用の switching network を持っているが，PC cluster はローカルエリアネットワークを用いているので，他の computer のメモリにアクセス (通信) するのは遅い。

3.2 結合ネットワーク

並列計算機ではプロセッサ間で相互結合ネットワークを用いて通信しなければならない。相互結合ネットワークには以下の二種類のものがある。

- Shared Media
  一時には一つのメッセージの通信しか許されない。すべてのプロセスは送られたメッセージが自分宛のものかどうかを判別する必要がある。Shared Media の最も一般的な例は Ethernet である。

- Switched Media
  point-to-point 通信ができる。いくつものメッセージ，例えば，A $\rightarrow$ B と C $\rightarrow$ D を，同時に送信できる。Mesh 型，Tree 型，hyper cube 型 など多くのものが開発された。

3.3 Flynn の分類

- SISD (Single Instruction stream and Single Data stream)
  通常の逐次計算機

- SIMD (Single Instruction stream and Multiple Data stream)
  Processor array

- MISD (Multiple Instruction stream and Single Data stream)
  Systolic array

- MIMD (Multiple Instruction stream and Multiple Data stream)
  Multiprocessor と Multicomputer

3.4 これからのスーパーコンピュータ

- ベクトル SMP 方式
  例 地球シミュレータ
  コストが高い，電力の問題。1 PFLOPS を目指すには，1 TFLOPS × 1,000 ノード程度。

- スカラ SMP 方式
  例 ASCI Purple, 富士通 Prime Power
  汎用プロセッサと汎用メモリを用いているのでコストは安いが，メモリ，ネットワーク性能の性能を高める必要がある。1 PFLOPS を目指すには，100 GFLOPS × 10,000 ノード程度。

- 超並列 方式例 BlueGene/L
  コストが安く省スペースだが，アプリケーションソフトウェアを超並列計算向きへ書き換える必要がある。1 PFLOPS を目指すには，10 GFLOPS × 100,000 ノード程度。

TOP 500 list の 100 以内に入っている日本製コンピュータの数：

年	台数	年	台数
1994	32	2000	27
1995	22	2001	26
1996	23	2002	18
1997	44	2003	16
1998	36	2004	6
1999	25	2005	6

課題 TOP 500 list に入っているコンピュータのどれか一つ選びを，そのアークテクチャの特徴を調べよ。

3.5 ベンチマークテストについて

- SPEC [19]
  Standard Performance Evaluation Corporaton (標準性能評価法人) が提供する benchmark プログラム。OS や ハードウェアに依存しない言語 (C, Fortran) で書かれている。SPECint, SPECfp 値で計算機の性能を評価する。

- Linpack [10]
  J. Dongarra, J.R. Bunch, C.B. Moler, G.W. Stewart 等が Fortran で開発した線形計算のサブルーチンライブラリである Linpack を用いて何 GFLOPS の性能を出したかで，ハードウェアの性能を評価する。科学技術計算向きのベンチマークテストであり，TOP 500 はこの値で順位が付けられる。

- 姫野ベンチマーク [7]
  理化学研究所 姫野龍太郎氏が作成した性能測定プログラムで，内容は流体解析計算のプログラム。国内の並列計算機メーカは姫野ベンチの結果を表示することが多い。

4　並列プログラムの性能評価

並列プログラムを実行するとき，実行する前に性能が予測できれば，実際に並列処理で行うべきか，逐次処理で実行すべきか，判断する材料が提供されたことになる。ここでは並列プログラムの性能評価を学ぶ。

4.1　速度向上比と効率

並列計算における速度向上比 (Speedup) S と 効率 (Efficiency) E とは

$$S = \frac{\text{逐次プログラムの実行時間}}{\text{並列プログラムの実行時間}}$$
$$E = \frac{S}{p}$$

によって定義される。ここで，p は使われているプロセッサの数である。この定義より

$$0 \leq S \leq p$$
$$0 \leq E \leq 1$$

を得る。

4.2　Amdahl の法則

並列処理プログラムの速度向上比，効率を解析し向上させるには，プログラムの中には，

1. どうしても逐次的に実行しなければならない部分
2. 並列に実行できる部分
3. 並列処理に付随して出てきた計算とは直接関係ない部分 (Parallel overhead)

の 3 つの部分があることを考慮しなければならない。

いま大きさ n の問題を解くとき，上に示した最初の部分の計算時間を $\sigma(n)$，二番目の並列処理可能な部分を逐次処理した場合の処理時間を $\varphi(n)$ とし，三番目の部分の計算時間を $\kappa(n,p)$ とする。そうすると，逐次処理での処理時間は $\sigma(n)+\varphi(n)$ であったものが，並列処理では $\sigma(n)+\varphi(n)/p+\kappa(n,p)$ となるので，速度向上比は

$$S(n,p) = \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

で与えられる。しかしこれは「並列に計算できる部分」が完全に並列化できた場合を想定しているので，実際は

$$S(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)} \tag{1}$$

となる。これより，効率 $E(n,p) = S(n,p)/p$ は

$$E(n,p) \leq \frac{\sigma(n)+\varphi(n)}{p\,\sigma(n)+\varphi(n)+p\,\kappa(n,p)}$$

となる。また，$\kappa(n,p) \geq 0$ であるから

$$S(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)} \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p}$$

を得る。

ここで，本質的に並列化不可能で逐次計算するしかない部分の処理時間の比率を f で表す。すなわち

$$f = \sigma(n)/(\sigma(n) + \varphi(n))$$

とすれば

$$\begin{aligned} S(n,p) &\leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p} \\ &= \frac{\sigma(n)/f}{\sigma(n) + \sigma(n)(1/f - 1)/p} \\ &= \frac{1/f}{1 + (1/f - 1)/p} \\ &= \frac{1}{f + (1 - f)/p} \end{aligned}$$

となる。これを **Amdahl の法則** という。

Amdahl の法則

$f(0 \leq f \leq 1)$ を並列プログラムの中の逐次処理部分の割合とすると，プロセッサ数 p で並列計算を行うと，速度向上比は

$$S(n,p) \leq \frac{1}{f + (1 - f)/p}$$

を満たす。

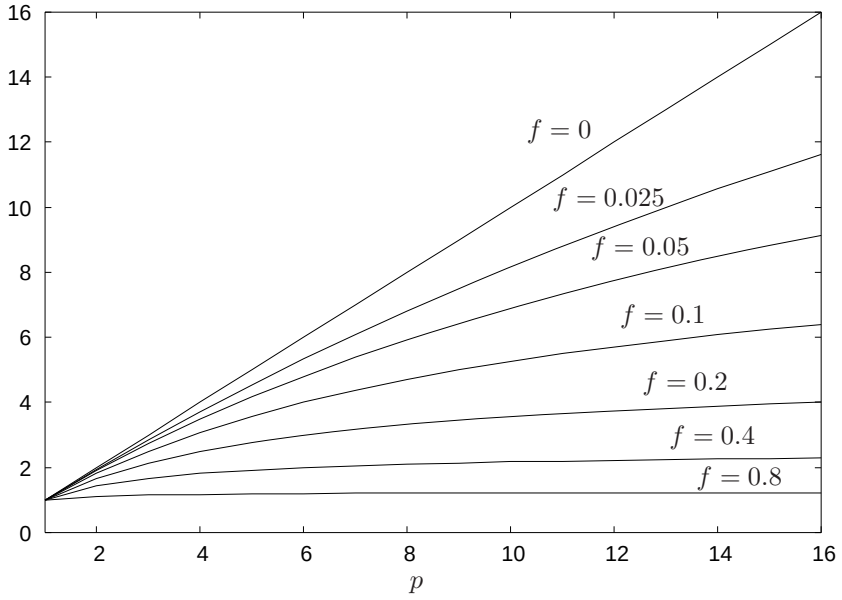
例 一つのプログラムの中で逐次処理を行わなければならない部分が 10% あったとする。すなわち $f = 0.1$ とすると，プロセッサ数 $p = 8$ のとき，

$$S \leq \frac{1}{0.1 + (1 - 0.1)/8} \approx 4.71$$

であるので，うまくいけばおおよそ 4.71 倍高速化できる。また，プロセッサ数 p を限りなく増やしていくと

$$\lim_{p \to \infty} \frac{1}{f + (1 - f)/p} = \frac{1}{f} = 10$$

となるので，せいぜい 10 倍の高速化しか得られない。ここで p と f をいくつか変えて S の変化の様子を図示する。



Amdahl の法則

4.3 Gustafson の法則

Amdahl の法則は

- 問題のサイズ n および逐次部分の割合 f を固定し，
- プロセッサ数 p を増やし

たとき，どれだけ計算時間が短縮されるかを示している。しかし，Amdahl の法則では

- 時間を固定し
- プロセッサ数とともに問題のサイズを大きくし

たとき，どれだけ高速化されるかはわからない。このような要求に応えてくれるのが **Gustafson の法則** である。

いま，問題のサイズを n とし，逐次部分の演算量が $O(n)$，並列部分の演算量を $O(n^2)$ とすると，n が大きくなるにつれ逐次部分の割合が小さくなっていく。並列計算を行ったときの逐次部分の割合を s とすると

$$s = \frac{\sigma(n)}{\sigma(n) + \varphi(n)/p}$$
$$1 - s = \frac{\varphi(n)/p}{\sigma(n) + \varphi(n)/p}$$

となる。これより

$$\sigma(n) = (\sigma(n) + \varphi(n)/p)\, s$$
$$\varphi(n) = (\sigma(n) + \varphi(n)/p)\, (1 - s)\, p$$

を得る。これを

$$S \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

に代入すると

$$S \leq s + (1 - s)\, p = p + (1 - p)\, s \tag{2}$$

を得る。

例 あるアプリケーションプログラムを実行しようとしている。このプログラムの計算時間の 5% が逐次部分で費されているとすると，64 台のプロセッサを使うと

$$S = 64 + (1 - 64) \times 0.05 = 60.85$$

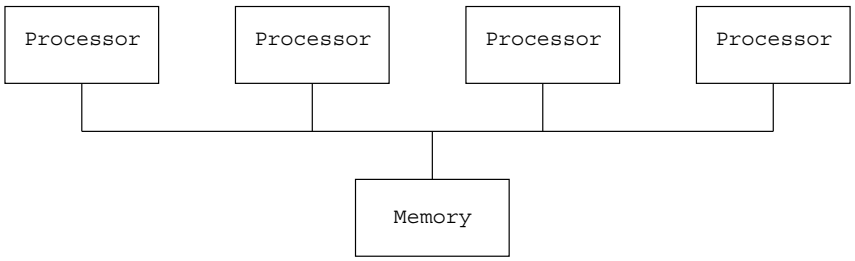
倍高速化される。

5　並列プログラミング

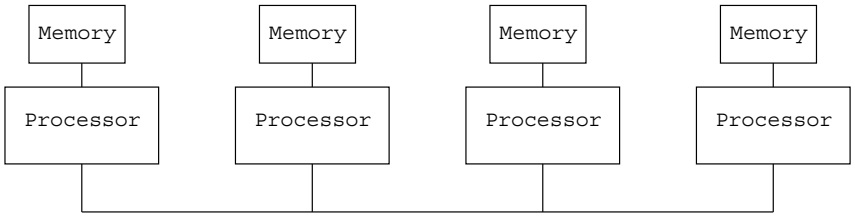
並列処理には，**スレッド 並列** と **プロセス並列** という 2 種類の処理方法がある。スレッドとはメモリを共有する処理の単位で，プロセスとはそれよりは大きいメモリを共有しない処理の単位である。それぞれの計算機アークテクチャとの関係は下の表のようになっている。

	共有メモリ型	分散メモリ型
並列処理の種類	スレッド 並列	プロセス並列
ソフトウェア	OpenMP	MPI

共有メモリ型並列計算機 とは，いくつかの Processor がメモリを共有している並列計算機を言い，**分散メモリ型並列計算機** とは，複数の独立した計算機がネットワークを介して結合しているものを言う。分散型の場合，他のプロセッサのメモリへは直接アクセスできず，ネットワークを介してアクセスする (下図参照)。

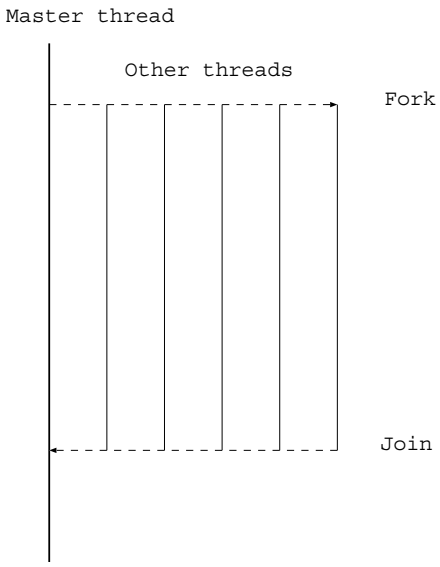


共有メモリ型並列計算機の概念図



分散メモリ型並列計算機の概念図

スレッド 並列では，プログラムは，開始時には **マスタースレッド** という 1 つのスレッドのみがアクティブで，並列化の命令に出会うと，いくつかのスレッドに分岐 (fork) し，以後，マスタースレッドと他のスレッドは同時に並列部分を実行する。並列部分の実行が終わると，マスタースレッド 以外の他のスレッド 消滅し，あるいは待機し，以後は一つのスレッドだけが走る (join)。一方，プロセス並列では，始めから特立なプロセスがいくつか生成されていて，それらが強調しながら動いていく。



Fork/Join について

共有メモリ型並列計算機と分散メモリ型並列計算機の特徴

	共有メモリ型	分散メモリ型
プログラミング	容易	やや難しい
スケーラビリティ	低い	高い
コンパイラ	OpenMP を理解する特殊なコンパイラと実行ライブラリが必要	特殊なコンパイラは不要で，MPI のライブラリが必要。

5.1 共有メモリプログラミング

5.2 分散メモリプログラミング

6 並列アルゴリズム

6.1 Recursive Doubling

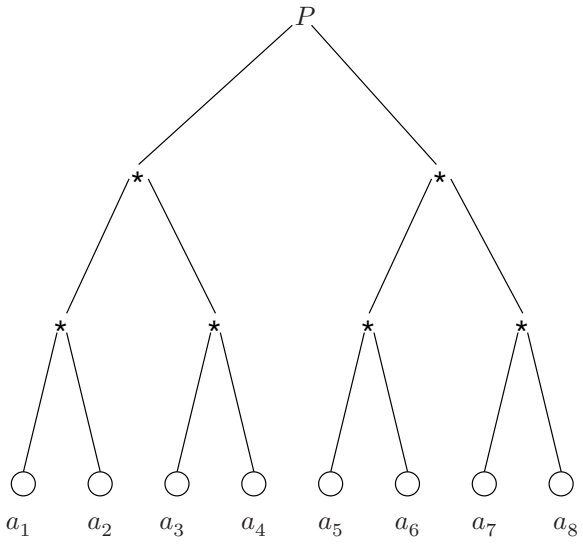
これは，n 個のデータ $a_1, a_2, \ldots, a_n$ に対して，結合則を満たす演算子 $\circ$ によって

$$P = a_1 \circ a_2 \circ \cdots \circ a_n$$

という演算を並列に計算するのに有効な手法である。例えば，8 個のデータ $a_1, \ldots, a_8$ の積 P は

$$\begin{aligned} P &= a_1 * a_2 * a_3 * a_4 * a_5 * a_6 * a_7 * a_8 \\ &= (a_1 * a_2 * a_3 * a_4) * (a_5 * a_6 * a_7 * a_8) \\ &= (a_1 * a_2) * (a_3 * a_4) * (a_5 * a_6) * (a_7 * a_8) \end{aligned} \tag{3}$$

であるから，



乗積 (3) の Recursive doubling による計算

という図にしたがって計算していけば，逐次計算では掛け算 7 回 (7 ステップ) 必要なところ，並列処理では 3 ステップで終了する。ただし，この場合，第 1 ステップでは 4 つのプロセッサが働き，第 2 ステップでは 2 つのプロセッサが働き，最終ステップでは 1 つしか働かないので効率は徐々に下っていく。なお，結合則を満たす演算子には

$$+,\ *,\ \max,\ \min$$

などがある。

6.2 並列ソーティング (バブルソーティングの場合)

いま n 個のデータ $a_i\,(i = 1, \ldots, n)$ があってこれを小さい順 (昇順) に並び替えることを考える。

バブルソート [12]

- 第 1 ステップ
  $i = 1, 2, \ldots, n-1$ について a_i と a_{i+1} を比較し，$a_i > a_{i+1}$ なら入れ替える。終了時には a_n が最大値になっている。

- 第 2 ステップ

  $i = 1, 2, \ldots, n-2$ について a_i と a_{i+1} を比較し，$a_i > a_{i+1}$ なら入れ替える。

$$\vdots$$

- 第 $(n-1)$ ステップ
  a_1 と a_2 を比較し，$a_1 > a_2$ なら入れ替える。

例 5個のデータ 8,4,7,6,1 をバブルソートで並び替える (太字が比較・交換を行った箇所)。

第 1 ステップ

	a_1	a_2	a_3	a_4	a_5
start	8	4	7	6	1
1	**4**	**8**	7	6	1
2	4	**7**	**8**	6	1
3	4	7	**6**	**8**	1
4	4	7	6	**1**	**8**

第 2 ステップ

	a_1	a_2	a_3	a_4	a_5
start	4	7	6	1	8
1	**4**	**7**	6	1	8
2	4	**6**	**7**	1	8
3	4	6	**1**	**7**	8

第 3 ステップ

	a_1	a_2	a_3	a_4	a_5
start	4	6	1	7	8
1	**4**	**6**	1	7	8
2	4	**1**	**6**	7	8

第 4 ステップ

	a_1	a_2	a_3	a_4	a_5
start	4	1	6	7	8
1	**1**	**4**	6	7	8

この方法では，合計

$$(n-1)+(n-2)+\cdots+2+1=\frac{n\,(n-1)}{2}$$

回の比較・交換が行われる。しかし複数のプロセッサがあれば同時にできる部分もある。例えば，第一ステップの a_1, a_2 の比較・交換と a_2, a_3 の比較・交換が終われば，直ちに第二ステップの a_1, a_2 の比較・交換が可能になるし，第一ステップの a_3, a_4 の比較・交換が終われば直ちに第二ステップの a_2, a_3 の比較・交換が可能になる (図 1 参照)。このようなことに注目すればバブルソートの並列化は容易にできる。逐次計算では，$n\,(n-1)/2$ ステップ要したものが，十分な数だけプロセッサがあれば，$2\,n-3$ ステップで終了する。

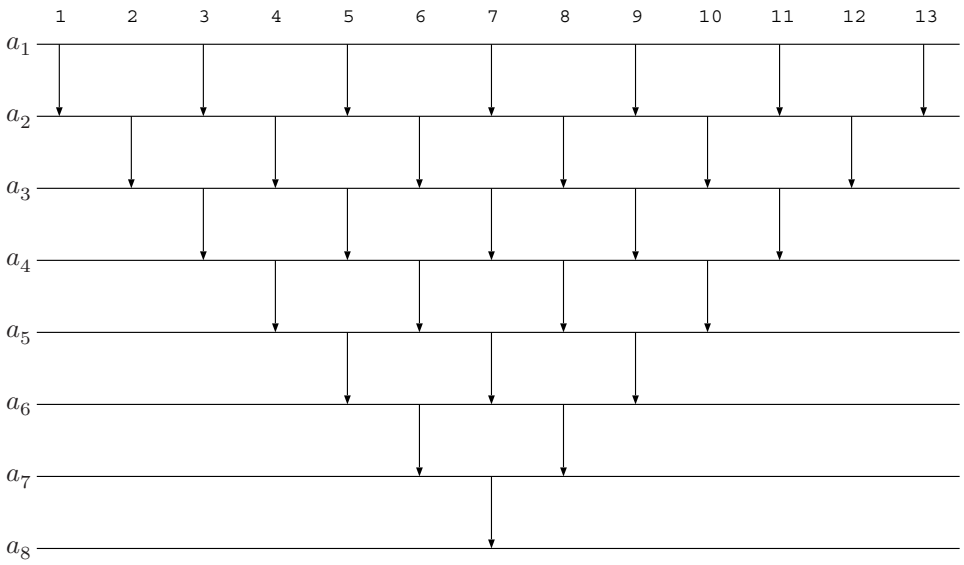


図 1. バブルソートの並列化 (↓ は比較・交換の処理を表している)。

7　並列プログラミング言語

各ベンダーが Fortran や C をベースに並列計算機向きの言語やライブラリを作成してきたが，これはポータビリティを考えると好ましいことではない。また，ハードウェアの進歩に伴って言語の仕様が頻繁に変わるという問題も生じてきた。そこでこれらの問題を解消すべく，1990 年代の始めのころから，並列プログラミングの API (Application Programming Interface) の標準化が行われるようになった。分散メモリプログラミングでは MPI (Message Passing Interface)，共有メモリプログラミングでは OpenMP が開発され，現在は発展途上である。

7.1　OpenMP について

現在のところ，共有メモリプログラミングの主流は，C や Fortran などの既存の言語の中で並列処理のためのライブラリを呼び出し，実行する方式が主流である。そのためのインターフェースの事実上の標準は OpenMP である。OpenMP の命令は，Fortran では **directive**，C および C++ では **pragma** という指示行を挿入することによって始まる。OpenMP の命令は，Fortran 90 では

```
!$omp ...
```

C では

```
#pragma omp ...
```

という書き出しで始まる。OpenMP は

1. directive+命令
2. Runtime library
3. 環境変数

から成っている。

プログラム例 1

```
1: Program hello
2: !  print *, "Hello parallel world from threads:"
3: !$omp parallel
4:   write(*,'(" Hello from thread",i2)') omp_get_thread_num()
5: !$omp end parallel
6: end Program hello
```

実行結果

```
Hello from thread 0
Hello from thread 3
Hello from thread 2
Hello from thread 1
```

プログラム例 2

```
1: Program do_1oop
2:   integer omp_get_thread_num,i,iam
3: !$omp parallel do private(iam)
```

```
    4:   do i=1,8
    5:      iam=omp_get_thread_num()
    6:      write (*,'(" Thread No. is",i3,", and i is",i3)') iam,i
    7:   end do
    8: !$omp end parallel do
    9: end Program do_loop
```

実行結果

```
Thread No. is  0, and i is  1
Thread No. is  0, and i is  2
Thread No. is  1, and i is  3
Thread No. is  1, and i is  4
Thread No. is  3, and i is  7
Thread No. is  3, and i is  8
Thread No. is  2, and i is  5
Thread No. is  2, and i is  6
```

7.2 MPI について

分散メモリ型並列計算機では，ある計算ノードが他計算ノードのメモリ上にあるデータを必要としたとき，通信によりデータを受け取る必要がある。UNIX のプログラミング環境において通信を行うとすると，ソケット通信と呼ばれるプログラミング手法を用いるのが一般的である。ソケット通信はもっとも基本的な通信手法であるが，プログラミングには TCP/IP の知識を必要とし，初心者には容易ではない。そこで並列計算機を製造している各ベンダーはプログラマが利用しやすい通信関数をそれぞれ独立に用意していた。しかしながらこれらの関数は統一された規格ではないので，異なるベンダーの並列計算機でプログラミングする場合には一からそのベンダの作成した通信ライブラリを勉強しなければならなかった。これでは不便であるので，1994 年に研究者やベンダらによって組織された Message Passing Interface Forum [14] により，並列計算用通信ライブラリの標準規格として Message Passing Interface(以下では MPI を略す) が定められた。MPI ライブラリを用いることにより，同一のプログラムによって異なるプラットフォーム上で並列計算を行うことが可能である。

MPI の実装には MPICH [13]，PVM [17] などがある。どちらを利用してもよいが，ここでは MPICH を利用するものとして話を進める。mpich のインストール方法は付録 9.1 を参照せよ。

以下では MPI ライブラリの簡単な使い方を説明する。

7.3 コンパイルと実行

簡単なプログラムを例に MPI を用いたプログラムのコンパイル方法と実行方法を説明する。プログラムは単一のプログラムを作成すればよく，SPMD モデルにしたがい同一のプログラムが全ノードで実行され，プロセスランクといわれる識別子によって各ノード (各プロセス) の作業が分岐する仕組みになっている。例として，プロセスランクが 0 のプロセスのみが標準出力に「Hello World!」と表示するプログラムをプログラムリスト 1 に与える。

プログラムリスト 1 (hello.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int my_rank;
 7
 8    MPI_Init(&argc,&argv);
 9    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
10
11    if (my_rank==0){
12      printf("Hello World!\n");
13    }
14
15    MPI_Finalize();
16    return 0;
17  }
```

まずプログラムリスト 1 のプログラムを解説する。MPI 関数を利用するためにはヘッダファイル mpi.h をインクルードしなければならない。上記のプログラムでは 2 行目でヘッダファイルをインクルードしている。

MPI 関数を呼び出す前には MPI_Init 関数を呼び出す必要があり，最後に MPI_Finalize 関数を呼び出す必要がある。上記のプログラムでは 7 行目と 15 行目でこれらの関数を呼び出している。

9 行目にある MPI_Comm_rank 関数はプロセスが自身のランクを知るための関数である。MPI_Comm_rank 関数の第 1 引数である MPI_COMM_WORLD は，並列計算に参加しているすべてのプロセスを示すコミュニケータである。通常は MPI_COMM_WORLD を指定しておけばよい。コミュニケータに関する詳しい説明は，パチェコ [16, pp.121–148] を参照せよ。

また MPI_Comm_rank 関数以外によく利用する MPI 関数として MPI_Comm_size 関数がある。この関数は並列計算の総プロセス数を調べる関数である。MPI を用いたプログラムでは必ず利用すると言ってもよい関数なので，覚えておく必要がある。

上記のプログラムをエディタで作成し，hello.c というファイル名で保存したとする。コンパイルと実行は以下のように行う。mpirun コマンドの np オプションは実行プロセス数を指定するオプションで，下記の例では 2 プロセスでプログラムが実行される。

```
$ mpicc hello.c -o hello
$ mpirun -np 2 hello
Hello World!
```

7.4 1 対 1 通信関数

もっとも単純な通信はデータの送信側と受信側が 1 対 1 に対応する 1 対 1 通信である。送信用の関数 MPI_Send と受信用の関数 MPI_Recv があり，これらを送信側と受信側がそれぞれ呼び出すことにより通信を行うことができる。これらの関数のプロトタイプ宣言は以下の通りである。

MPI_Send 関数

```
int MPI_Send(
 void*         message,
 int           count,
 MPI_Datatype  datatype,
 int           dest,
 int           tag,
 MPI_Comm      comm);
```

MPI_Recv 関数

```
int MPI_Recv(
 void*         message,
 int           count,
 MPI_Datatype  datatype,
 int           source,
 int           tag,
 MPI_Comm      comm,
 MPI_Status*   status);
```

まず MPI_Send 関数の引数について説明する。第 1 引数は送信したメッセージを格納した変数または配列を指定する。第 2 引数は送信メッセージの個数を指定する。第 3 引数は送信するデータの型を指定する。MPI で利用できる型と C 言語におけるデータ型との対応を表 1 に与える。第 4 引数は送信先のランクを指定する。第 5 引数はタグを指定する。プログラマは通信終了後の処理を区別する目的でタグを利用することができる。例えば，タグが 0 の受信データはストア用で，タグが 1 の受信データは出力用といった使い方が可能である。第 6 引数はコミュニケータを指定する。通常は MPI_COMM_WORLD を指定すればよい。例えば，int 型の配列 a を先頭アドレスから 5 つ分ランク 0 に送信する場合は以下のように関数を呼び出せばよい。

```
MPI_Send(a,5,MPI_INT,0,0,MPI_COMM_WORLD);
```

表 1: MPI で利用できるデータ型

MPI のデータ型	C 言語のデータ型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

次に MPI_Recv 関数の引数を説明する。第 1 引数から第 3 引数は MPI_Send 関数と同様である。第 4 引数は送信元のプロセスのランクを指定する。第 5，第 6 引数は MPI_Send 関数と同様である。第 7 引数は，実際に受信されたデータに関する情報 (送信元，タグ，エラーコードなど) を保存する MPI_Status 型の構造体を指定する。例えば，int 型データを 5 つ分ランク 1 から受信し，int 型の配列 b に保存する場合は以下のように関数を呼び出せばよい。

```
MPI_Recv(b,5,MPI_INT,1,0,MPI_COMM_WORLD,status);
```

ランク 1 で定義された int 型配列 a のデータを 5 つ分をランク 0 に送信し，データを受信したランク 0 のプロセスはそのデータを配列 b に保存し，標準出力に表示するプログラムをプログラムリスト 2 に与える。

プログラムリスト 2 (send.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int i;
 7    int my_rank;
 8    int a[5],b[5];
 9    MPI_Status status;
10
11    for(i=0;i<5;i++){
12      a[i]=i;
13    }
14
15    MPI_Init(&argc,&argv);
16    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
17
18    if (my_rank==0){
19      MPI_Recv(b,5,MPI_INT,1,0,MPI_COMM_WORLD,&status);
20    }
21    else if (my_rank==1){
22      MPI_Send(a,5,MPI_INT,0,0,MPI_COMM_WORLD);
23    }
24
25    if (my_rank==0){
26      for(i=0;i<5;i++){
27        printf("b[%d]=%d\n",i,b[i]);
28      }
29    }
30
31    MPI_Finalize();
32    return 0;
33  }
```

send.c の実行結果

```
$ mpicc send.c -o send
$ mpirun -np 2 send
b[0]=0
b[1]=1
b[2]=2
b[3]=3
b[4]=4
```

7.5 集団通信関数

前節で 1 対 1 通信を学んだ。基本的にすべての通信を 1 対 1 通信を用いて行うことができるが，規模の大きな並列計算システムではプロセス数も大きくなり，その場合に一つ一つの通信を 1 対 1 通信で行おうとすると関数の呼び出し回数が増え，プログラムが複雑になってしまう。また，このとき効率よく通信を行うことに十分な注意を払ってプログラミングしないと効率が落ちてしまうという問題もある。MPI はこういった問題を気にすることなく利用することができる集団通信関数を用意している。ここでは，集団通信関数 MPI_Bcast，MPI_Reduce，MPI_Allreduce，MPI_Gather，MPI_Scatter，MPI_Allgather 関数を紹介する。

7.5.1 MPI_Bcast 関数

MPI_Bcast 関数はあるランクのプロセスがもっているデータを他のすべてのプロセスに送信する関数である。この関数のプロトタイプ宣言は以下の通りである。

MPI_Bcast 関数

```
int MPI_Bcast(
 void*         message,
 int           count,
 MPI_Datatype  datatype,
 int           root,
 MPI_Comm      comm);
```

第 4 引数では送信元のプロセスランクを指定する。他の引数は MPI_Send 関数の場合と同様であるので説明は省略する。MPI_Bcast 関数が呼び出された後，第 1 引数で指定された変数 (または配列) はすべてのプロセスで同一のデータを持つことになる。例えば，ランク 0 の int 型変数 a を他のすべてのプロセスに送信する場合は以下のように MPI_Bcast 関数を呼び出せば良い。この関数を呼び出した後，変数 a の値はどのプロセスでも同じになる。

```
MPI_Bcast(a,1,MPI_INT,0,MPI_COMM_WORLD);
```

MPI_Bcast 関数を用いて，ランク 0 で int 型変数 a に保存したデータをすべてのプロセスに送信するプログラムをプログラムリスト 3 に与える。また実行結果は以下の通りである。

プログラムリスト 3 (bcast.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int my_rank,p;
 7    int i,a=0;
 8
 9    MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12
13    if (my_rank==0){
14      a=10;
15    }
16
17    printf("送信前: my_rank=%d a=%d\n",my_rank,a);
18
19    MPI_Bcast(&a,1,MPI_INT,0,MPI_COMM_WORLD);
20
21    printf("送信後: my_rank=%d a=%d\n",my_rank,a);
22
23    MPI_Finalize();
24
25    return 0;
26  }
```

bcast.c の実行結果

```
$ mpicc bcast.c -o bcast
$ mpirun -np 2 bcast
送信前: my_rank=0 a=10
送信後: my_rank=0 a=10
送信前: my_rank=1 a=0
送信後: my_rank=1 a=10

$ mpirun -np 4 bcast
送信前: my_rank=0 a=10
送信後: my_rank=0 a=10
送信前: my_rank=1 a=0
送信後: my_rank=1 a=10
送信前: my_rank=2 a=0
送信後: my_rank=2 a=10
送信前: my_rank=3 a=0
送信後: my_rank=3 a=10
```

7.5.2 MPI_Gather

MPI_Gather 関数は，各プロセスに分散しているデータを 1ヶ所に集める関数である (図 1)。この関数のプロトタイプ宣言は以下の通りである。

MPI_Gather 関数

```
int MPI_Gather(
 void*         send_data,
 int           send_count,
 MPI_Datatype  send_type,
 void*         recv_data,
 int           recv_count,
 MPI_Datatype  recv_type,
 int           root,
 MPI_Comm      comm);
```

MPI_Gather 関数の引数を説明する。第 1 から第 3 引数は送信されるデータを指定する部分で，送信したいデータが格納されている変数 (または配列) を第 1 引数に指定する。第 2 引数にでは，送信したいデータの個数を指定する。第 3 引数は送信したいデータのデータ型を指定する。第 4 から第 6 引数は受信データに関する指定で，第 4 引数に受信したデータを格納する配列を指定する。第 5 引数と第 6 引数は第 2，第 3 引数と同様に設定する。第 7 引数はデータを集めてくるプロセスのランクを指定する。第 8 引数はコミュニケータを指定する。例えば，各プロセスで別々に定義された int 型変数 a をランク 0 の配列 b に集めてくるには以下のように MPI_Gather 関数を呼び出せば良い。

```
MPI_Gather(&a,1,MPI_INT,b,1,MPI_INT,0,MPI_COMM_WORLD);
```

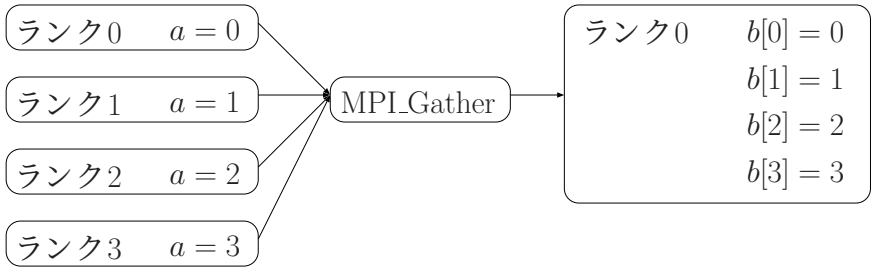


図 1: MPI_Gather 関数

プログラム例として，各プロセスが変数 a に自分のランク数を保存し，それらをランク 0 に MPI_Gather 関数で集めてくるプログラムをプログラムリスト 4 に与える。

プログラムリスト 4 (gather.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int my_rank,p;
 7    int i,a,b[10];
 8
 9    MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &p);
13
14    a=my_rank;
15
16    MPI_Gather(&a,1,MPI_INT,b,1,MPI_INT,0,MPI_COMM_WORLD);
17
18    if(my_rank==0){
19      for(i=0;i<p;i++)
20        printf("b[%d]=%d\n",i,b[i]);
21    }
22
23    MPI_Finalize();
24
25    return 0;
26  }
```

gather.c の実行結果

```
$ mpicc gather.c -o gather
$ mpirun -np 2 gather
b[0]=0
b[1]=1

$ mpirun -np 4 gather
b[0]=0
b[1]=1
b[2]=2
b[3]=3
```

7.5.3 MPI_Scatter

MPI_Gather 関数は各プロセスに散らばっているデータを集めてくる関数であった。これの逆，すなわち，1箇所にあるデータを各プロセスに分散させる関数が MPI_Scatter 関数である (図 2)。この関数のプロトタイプ宣言は以下の通りである。引数は MPI_Gather 関数と同様であるので説明は省略する。

MPI_Scatter 関数

```
int MPI_Scatter(
 void*         send_data,
 int           send_count,
 MPI_Datatype  send_type,
 void*         recv_data,
 int           recv_count,
 MPI_Datatype  recv_type,
 int           root,
 MPI_Comm      comm);
```

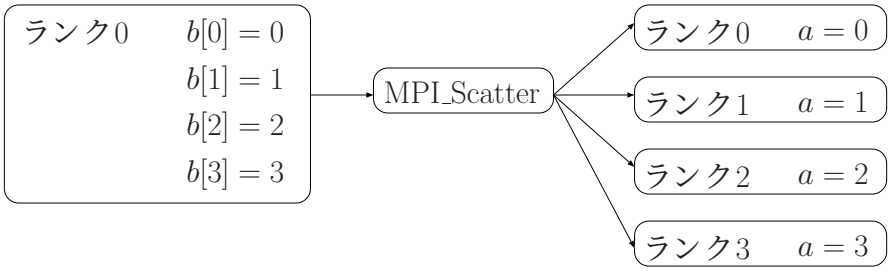


図 2: MPI_Scatter 関数

ランク 0 で定義された int 型配列 b を各プロセスにデータを 1 つずつ分配するには以下のように MPI_Scatter 関数を呼び出せば良い。

```
MPI_Scatter(b,1,MPI_INT,&a,1,MPI_INT,0,MPI_COMM_WORLD);
```

プログラム例として，ランク 0 のプロセスが配列 b に適当なデータを持っていて，それらを各プロセスに MPI_Scatter 関数によって分配させるプログラムをプログラムリスト 5 に与える。

プログラムリスト 5 (scatter.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int my_rank,p;
 7    int i,a,b[10];
 8
 9    MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &p);
13
14    if(my_rank==0){
15      for(i=0;i<p;i++)
16        b[i]=i;
17    }
18
19    MPI_Scatter(b,1,MPI_INT,&a,1,MPI_INT,0,MPI_COMM_WORLD);
20
21    printf("my_rank=%d a=%d\n",my_rank,a);
22
23    MPI_Finalize();
24
25    return 0;
26  }
```

scatter.c の実行結果

```
$ mpicc scatter.c -o scatter
$ mpirun -np 2 scatter
my_rank=0 a=0
my_rank=1 a=1

$ mpirun -np 4 scatter
my_rank=0 a=0
my_rank=1 a=1
my_rank=2 a=2
my_rank=3 a=3
```

7.5.4 MPI_Allgather

MPI_Gather 関数では一つのプロセスにデータを集めたが，集めたデータがすべてのプロセスで必要になる場合もある。その場合は MPI_Gather 関数で集めてきたデータを MPI_Bcast 関数ですべてのプロセスに送信するということも考えられるが，MPI_Allgather 関数を用いる方が効率的である (図 3)。この関数のプロトタイプ宣言は以下の通りである。

MPI_Allgather 関数

```
int MPI_Allgather(
 void*         send_data,
 int           send_count,
 MPI_Datatype  send_type,
 void*         recv_data,
 int           recv_count,
 MPI_Datatype  recv_type,
 MPI_Comm      comm);
```

引数は MPI_Gather 関数とほとんど同じなので，説明は省略する。例えば，各プロセスで別々に定義された int 型変数 a をすべてのプロセスの配列 b に集めてくるには以下のように MPI_Allgather 関数を呼び出せば良い。

```
MPI_Allgather(&a,1,MPI_INT,b,1,MPI_INT,MPI_COMM_WORLD);
```

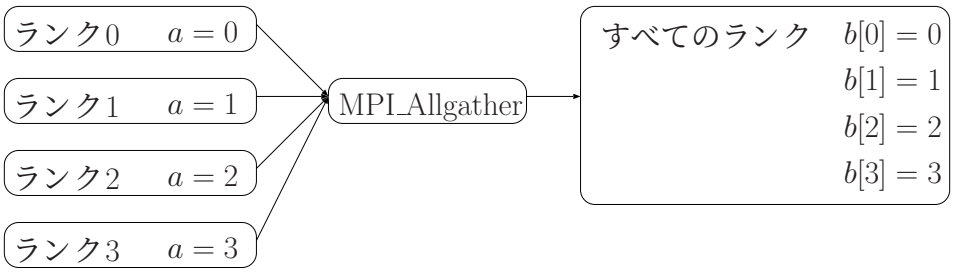


図 3: MPI_Allgather 関数

プログラム例として，各プロセスが変数 a に自分のランク数を保存し，それらを MPI_Allgather 関数で全プロセスに集めてくるプログラムをプログラムリスト 6 に与える。

プログラムリスト 6 (allgather.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int my_rank,p;
 7    int i,a,b[10];
 8
 9    MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12    MPI_Comm_size(MPI_COMM_WORLD, &p);
13
14    a=my_rank;
15
16    MPI_Allgather(&a,1,MPI_INT,b,1,MPI_INT,MPI_COMM_WORLD);
17
18    for(i=0;i<p;i++)
19      printf("my_rank=%d b[%d]=%d\n",my_rank,i,b[i]);
20
21    MPI_Finalize();
22
23    return 0;
24  }
```

allgather.c の実行結果

```
$ mpicc allgather.c -o allgather
$ mpirun -np 2 allgather
my_rank=0 b[0]=0
my_rank=0 b[1]=1
my_rank=1 b[0]=0
my_rank=1 b[1]=1

$ mpirun -np 4 allgather
my_rank=0 b[0]=0
my_rank=0 b[1]=1
my_rank=0 b[2]=2
my_rank=0 b[3]=3
my_rank=1 b[0]=0
my_rank=1 b[1]=1
my_rank=1 b[2]=2
my_rank=1 b[3]=3
my_rank=2 b[0]=0
my_rank=2 b[1]=1
my_rank=2 b[2]=2
my_rank=2 b[3]=3
my_rank=3 b[0]=0
my_rank=3 b[1]=1
my_rank=3 b[2]=2
my_rank=3 b[3]=3
```

7.5.5 MPI_Reduce

並列計算を行うとき，プログラマは各プロセスに保存されているデータの中から最大の要素を見つけたいとかデータの総和を求めたいということに良く出くわす。これまで学習してきた知識でこれを実行しようと思えば，MPI_Gather 関数でデータを 1 箇所に集めてきて，その後最大値を計算したり，和を計算したりすればよいが，そうするより効率的に方法として MPI_Reduce 関数を使う方法がある。MPI_Reduce 関数はデータを通信によって集めてくるだけでなく，同時に和や最大値を求める操作 (リダクション操作と呼ばれる) を行う関数である (図 4)。この関数のプロトタイプ宣言は以下の通りである。

MPI_Reduce 関数

```
int MPI_Reduce(
 void*         operand,
 void*         result,
 int           conut,
 MPI_Datatype  datatype,
 MPI_Op        operator,
 int           root,
 MPI_Comm      comm);
```

MPI_Reduce 関数の引数を説明する。第 1 引数にはリダクション操作を行いたいデータが格納されている変数または配列を指定する。第 2 引数には結果を保存する変数または配列を指定する。第 3 引数にはデータの

個数を，第 4 引数には扱うデータの型を指定する。第 5 引数はどのようなリダクション操作を行うのかを指定する箇所で，表 2 に挙げたリダクション操作を選択することができる。第 6 引数は結果を保存するプロセスのランクを指定する。第 7 引数はコミュニケータを指定する。例えば各プロセスで int 型変数 a が定義されていて，それぞれ別々の値が保存されていたとする。これらの和をプロセスランク 0 の変数 sum に保存したい場合は，以下のように MPI_Reduce 関数を呼び出せば良い。

```
MPI_Reduce(&a,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
```

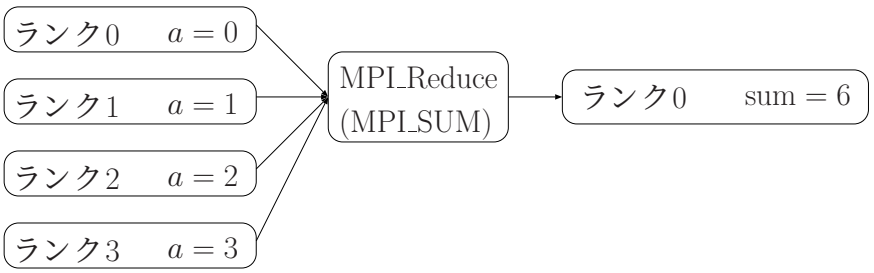


図 4: MPI_Reduce 関数

表 2: リダクション操作の種類

操作名	意味
MPI_MAX	最大値
MPI_MIN	最小値
MPI_SUM	和
MPI_PROD	積
MPI_LAND	論理積
MPI_BAND	ビット論理積
MPI_LOR	論理和
MPI_BOR	ビット論理和
MPI_LXOR	排他的論理和
MPI_BXOR	ビット排他的論理和

プログラム例として，以下を考える。各プロセスが変数 a に自分のランク数を保存し，それらの和を MPI_Reduce 関数で求め，その結果をランク 0 の変数 sum に保存するプログラムをプログラムリスト 7 に与える。

プログラムリスト 7 (reduce.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int my_rank;
 7    int a,sum;
 8
 9    MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12
13    a=my_rank;
14
15    MPI_Reduce(&a,&sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
16
17    if(my_rank==0){
18      printf("sum=%d\n",sum);
19    }
20
21    MPI_Finalize();
22
23    return 0;
24  }
```

reduce.c の実行結果

```
$ mpicc reduce.c -o reduce
$ mpirun -np 2 reduce
sum=1

$ mpirun -np 4 reduce
sum=6
```

7.5.6 MPI_Allreduce

MPI_Reduce 関数ではリダクション操作の結果は指定したランクのプロセスでしか利用できない。しかしながら，結果をすべてのプロセスで利用しなければならない場合も多い。そのような場合のために MPI_Allreduce 関数が用意されている。この関数のプロトタイプ宣言は以下の通りである。引数の説明は MPI_Reduce 関数と同様なので省略する。

MPI_Allreduce 関数

```
int MPI_Allreduce(
 void*         operand,
 void*         result,
 int           conut,
 MPI_Datatype  datatype,
 MPI_Op        operator,
 MPI_Comm      comm);
```

プログラム例として，各プロセスが変数 a に自分のランク数を保存し，それらの和を全プロセスの変数 sum に保存するプログラムをプログラムリスト 8 に与える。

プログラムリスト 8 (allreduce.c)

```
 1  #include <stdio.h>
 2  #include <mpi.h>
 3
 4  int main(int argc, char **argv)
 5  {
 6    int my_rank;
 7    int a,sum;
 8
 9    MPI_Init(&argc, &argv);
10
11    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
12
13    a=my_rank;
14
15    MPI_Allreduce(&a,&sum,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
16
17    printf("my_rank=%d sum=%d\n",my_rank,sum);
18
19    MPI_Finalize();
20
21    return 0;
22  }
```

reduce.c の実行結果

```
$ mpicc reduce.c -o reduce
$ mpirun -np 2 reduce
my_rank=0 sum=1
my_rank=1 sum=1

$ mpirun -np 4 reduce
my_rank=0 sum=6
my_rank=1 sum=6
my_rank=2 sum=6
my_rank=3 sum=6
```

7.6 Data 依存について

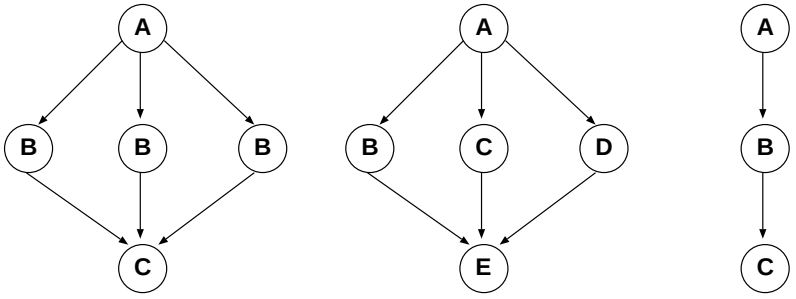
並列計算を行うには，同時性を検出しなければならない。

1. データ並列
   同じ演算 + を異なるデータに同時に適用可能。
   1: **for** $i = 1$ to 100 **do**
   2: a[i]=b[i]+c[i]
   3: **end for**

2. 機能並列
   3 行目と 4 行目が同時に実行できる。
   1: $a = 2$
   2: $b = 3$
   3: $m = (a + b)/2$
   4: $s = (a^2 + b^2)/2$
   5: $v = s - m^2$

並列化可能かどうかは **データ依存グラフ** によってわかる。

データ依存グラフ



データ並列　　機能並列　　逐次計算○はタスクを表し，その中の文字は実行されている演算を表す。矢印はタスク間の依存性を表す。

ある文においてデータが書き込まれたアドレスに，別な文でそこから読み込むとき，そのアドレスに **データ依存** があると言う。このデータ依存は並列化を妨げる要因になり得る。例えば，

```
1:  a=1; b=2; c=3
2:  a=a+b;
3:  c=c+a;
```

というプログラムを実行した結果，a=3, c=6 となるが，2 行目と 3 行目を逆に実行すると a=3, c=4 となり，意図した結果とは異なる。通常は，逆に実行することはないが，ループにおいて異なる index の値を別なスレッドあるいはプロセスで実行すると順序が逆になることもあり得る。

- ループ依存 (loop-carried dependence):

```
do i=2, n
  x(i) = x(i)+x(i-1)
end do
```

このプログラムを実行すると
i=2: x(2) ← x(2)+x(1)
i=3: x(3) ← x(3)+x(2)
となり i=2 の処理と i=3 の処理を別なスレッドで同時にできない。

- 非ループ依存 (non-loop-carried dependence):

```
do i=1,n
  px=y(i)+z(i)
  x(i)=px*x(i)
end do
```

このループを並列に実行すると px の値はスレッドごとに異なる。この場合は，変数 px を各スレッドのプライベート変数とすることによってデータ依存が除去できる。

7.6.1 データ依存の除去

文 S_1 の後に文 S_2 を実行し，それぞれメモリの同じ場所に Read(R) と Write(W) が実行されるとき，以下のような依存関係が生じ並列化を妨げることがある。

各種のデータ依存

順序	文	フロー依存	反依存	出力依存	依存性なし
↓	S_1	**W**	**R**	**W**	**R**
	S_2	**R**	**W**	**W**	**R**

1. 反依存とその除去法
予めデータのコピーを作っておけば並列に実行できる。

```
do i=1, n-1
  a(i) = a(i)+a(i+1)
end do
```

⇓

```
do i=1, n-1
  a2(i) = a(i+1)
end do
do i=1, n-1
  a(i) = a(i)+a2(i)
end do
```

2. 出力依存とその除去法

```
do i=2, n-1
  d = f(i-1)+f(i+1)
  fn(i) = f(i)+d
end do
e=d*dt
```

OpenMP では変数 d を 'lastprivate' に指定すれば並列化可能。

3. ループ依存の除去法

```
   do i=3, n
     x(i) = a*x(i-2)+b
   end do
      ⇓
!$omp parallel sections
!$omp section
   do i=4, n, 2
     x(i) = a*x(i-2)+b
   end do
!$omp section
   do i=3, n, 2
     x(i) = a*x(i-2)+b
   end do
```

このようにすれば一つの配列要素 (address) に他のスレッドが書き込んだり読み込んだりしなくなる。

7.6.2 多重ループにおけるデータ依存性とその除去

下のプログラムは，j 方向には依存性があるが，i 方向には依存性がないので i-loop は並列化できる。

```
 do j=2, n
   do i=1, n
     x(i, j)=x(i, j)+x(i, j-1)
   end do
 end do
```

したがって，

```
   do j=2, n
!$omp parallel do
     do i=1,n
       x(i, j)=x(i, j)+x(i, j-1)
     end do
   end do
```

とすれば並列化できる。しかし，j が変わる毎に fork-join および反復演算の割り当てが起こり，期待したほど速くならない可能性がある。そこで i-loop と j-loop を入れ替えて

```
!$omp parallel do
   do i=1, n
     do j=2, n
       x(i, j)=x(i, j)+x(i, j-1)
     end do
  end do
```

とすればよい。だがこのプログラムも問題がある。というのは，Fortran では，二次元配列は第一添字の方向 (i 方向) に address が連続しているので，j-loop は不連続なアクセスになりキャッシュ効率が下がる。そこで下のプログラムのように各スレッド毎に i-loop の分担を決めておく。

```
!$omp parallel private (id)
   id=omp_get_thread()
   do j=2, n
     do i=is(id), ie(id)
       x(i, j) = x(i, j)+x(i, j-1)
     end do
   end do
```

ところで，行列積 $A \times B$ の計算プログラムは

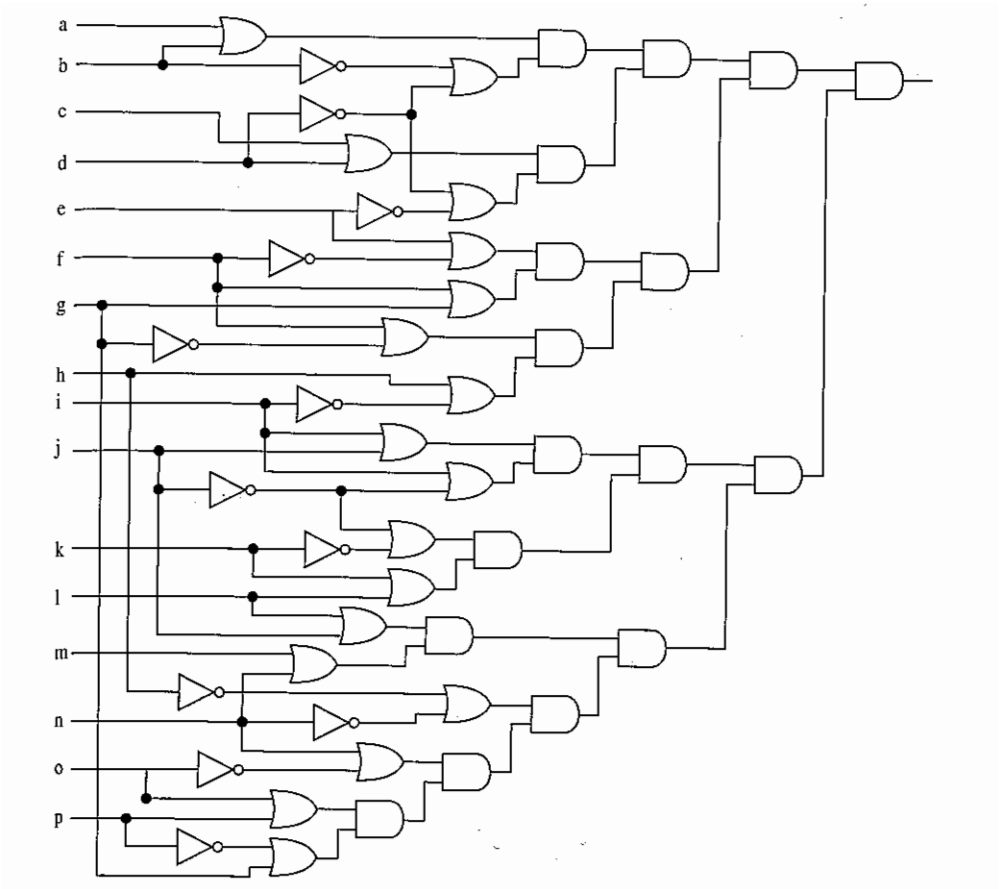
```
   do j=1, n
     do i = 1, n
       c(i, j)=0
       do k = 1, n
         c(i, j)=c(i, j)+a(i, k)*b(k, j)
       end do
     end do
   end do
```

となる。このプログラムの j-loop では，c(i, j) の一つの要素にしか access していないので，スレッドにまたがっての並列化は可能である。

8 応用例

8.1 Circuit Satisfiability 問題

図に示すような論理回路の入力端子 a–p に可能なあらゆる組み合わせの値を入力し，その中から，出力が $'1'$ となる入力を求める問題は，**Circuit satisfiability 問題** と呼ばれている。この答えを得るためには，ただ単に 16 個の入力端子 a–p に，$'000\cdots00'$ から $'111\cdots11'$ までの $2^{16} = 65,536$ 通りの値を入力し，その全てに対して出力が $'1'$ になるかどうかを調べればよい。その際，2^{16} 個の入力を，逐次的 (順番) に調べる必要はまったくな，それだけでなく，異なる入力に対する計算を行っている間には通信は不要だし，どの入力に対しても計算量がまったく等しいという究極の並列性を持っている。プロセッサが 2^{16} 個あれば 2^{16} 倍高速化されるはずである。



Circuit satisfiability 問題 ([11] より転載).

8.2 モンテカルロ法によるシミュレーション

計算機上で発生させる乱数は，ある「アルゴリズム」のもとで生成されている。そのため **疑似乱数** (pseudo random number) と呼んでいる。その中で最も代表的な **乗算合同法** (linear congrunetial method) のアルゴリズムとその並列化について学ぶ。

8.2.1 合同式の性質

乗算合同法を理解するためには合同式を理解しなければならない。整数 a と b が，整数 m で割ったとき余りが等しい，という関係にあれば

$$a \equiv b \pmod{m} \tag{4}$$

と書く。これを **a と b は m を法として合同である** と読む。このことは，$a-b$ が m の倍数である (m で割り切れる) ということ等価である。式 (4) のような式を合同式という。ここで，特に右辺の値 b が $0 \leq b < m$ を満たすならば（ようするに a を m で割った余りならば）

$$a = b \pmod{m} \tag{5}$$

と表すことにする。以下合同式の性質について説明する。

まず，二つの式

$$\begin{aligned} a &\equiv b \pmod{m} \\ c &\equiv d \pmod{m} \end{aligned} \tag{6}$$

が成り立っているとき

$$\begin{aligned} a \pm c &\equiv b \pm d \pmod{m} \\ a\,c &\equiv b\,d \pmod{m} \end{aligned} \tag{7}$$

が成り立つ。特に $c \equiv c \pmod{m}$ であるから

$$\begin{aligned} a \pm c &\equiv b \pm c \pmod{m} \\ a\,c &\equiv b\,c \pmod{m} \end{aligned} \tag{8}$$

となる。また，$a\,n \equiv b\,n \pmod{m}$ が成り立っていて m と n が互いに素のとき，両辺を n を割った式

$$a \equiv b \pmod{m} \tag{9}$$

が成り立つ。ようするに，合同式は等式と同じように，両辺に同じ数を書けたり足したり（割り算以外は）自由にできるということである。

8.2.2 乗算合同法とその並列化

乗算合同法のアルゴリズムは

$$x_{i+1} = a\,x_i \pmod{m}, \quad i = 0, 1, \ldots, \tag{10}$$

で与えられる。この式の意味するところは，a に x_i を掛け，それを m で割った余りを x_{i+1} とするということであるから，x_i が得られなければ x_{i+1} も得られない。したがって，このままでは x_i と x_{i+1} を同時に計算することはできない（並列計算ができない）。そこで，このアルゴリズムを変形して並列計算が可能なようにする。

式 (10) は合同式の特別な場合であるから，上で述べた合同式の性質から両辺に a を掛けても成り立つ。したがって，x_2 以降は

$$\begin{aligned} x_2 &= a\,x_1 = a^2\,x_0 \pmod{m} \\ x_3 &= a\,x_2 = a^2\,x_1 = a^3\,x_0 \pmod{m} \\ &\cdots \\ x_j &= a\,x_{j-1} = a^2\,x_{j-2} = \cdots = a^j\,x_0 \pmod{m} \end{aligned} \tag{11}$$

となる。ここで a^j を m で割った余りを α_j と表す。すなわち

$$a^j = \alpha_j \pmod{m}$$

とする。この式に $x_0 \equiv x_0 \pmod{m}$ を辺々掛けることにより

$$x_j = a^j\,x_0 = \alpha_j\,x_0 \pmod{m} \tag{12}$$

を得る。したがって，前もって α_j を求めておけば，この式を用いて一気に x_j が計算できる。以下，この式を用いて並列計算を行うことを考える。

8.2.3 Leapfrog 法

いま，p 個のプロセッサ $P_0, P_1, \ldots, P_{p-1}$ を用いて，式 (10) によって得られる乱数を並列に n 個発生させることを考える。ただし，n は p で割り切れるものと仮定し，各プロセッサにおいて下の表の割り当てに従って乱数を n/p 個発生させる。そのためには，各プロセッサのための種（表中太字の部分）を計算し，乗数を（a ではなく）a^p を m で割った値にする必要がある。

プロセッサ	乱 数				
P_0	$\boldsymbol{x_0}$	x_p	x_{2p}	$\cdots$	x_{n-p}
P_1	$\boldsymbol{x_1}$	x_{p+1}	x_{2p+1}	$\cdots$	x_{n-p+1}
P_2	$\boldsymbol{x_2}$	x_{p+2}	x_{2p+2}	$\cdots$	x_{n-p+2}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
P_i	$\boldsymbol{x_i}$	x_{p+i}	x_{2p+i}	$\cdots$	x_{n-p+i}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
P_{p-1}	$\boldsymbol{x_{p-1}}$	x_{2p-1}	x_{3p-1}	$\cdots$	x_{n-1}

以下 Leapfrog 法のアルゴリズムを示す。また OpenMP を用いた並列プログラムを付録に示しておく。

Leapfrog 法の並列計算

1. P_0 のための “種” $x_0 \neq 0$ を決める。
2. 他のプロセッサのための “種” $x_1, x_2, \cdots, x_{p-1}$ を

$$x_j = a\, x_{j-1} \qquad (\mathrm{mod}\ m), \quad j = 1, \ldots, p-1$$

より生成する。

3. $a^p = \alpha \pmod m$ を計算する。
4. プロセッサ $P_i\,(i = 0, 1, \ldots, p-1)$ にて $z_0 = x_i$ とし，漸化式

$$z_j = \alpha\, z_{j-1} \qquad (\mathrm{mod}\ m), \qquad j = 1, \ldots, \frac{n}{p} - 1$$

より乱数を生成する。

注意

- 上のアルゴリズムで得られた乱数列は m 未満の整数値であるから，区間 $(0, 1)$ の実数値に変換するためには m で割る必要がある。
- $a^p = \alpha \pmod m$ を求めるときは，a^p を計算してから余りを計算するのではなく，以下のような計算法で求める。

  1. $\alpha = 1$
     以下の演算を p 回繰り返す。
  2. $\alpha = a\,\alpha \pmod m$

- 一般的に m の値を大きくとる方が乱数列の周期は長くなる。通常の計算機では，整数値は 32bit の二進数で実現されていて，先頭の桁は符号を表しているから，取り得る m の最大値は 2^{31} である。$m = 2^{31}$ としたとき，m で割った余りを計算するには，31 bit 目（先頭桁）を 0 に置き換え，それ以下はそのまま残しておけばよいから，定数

$$2^{31} - 1 = 2147483647 = (011\cdots1)_2 = (7\mathrm{FFFFFFF})_{\mathrm{H}}$$

との bit 毎の論理積を取ればよい (Fortran では iand 関数，C では & 演算子を用いる)。

- 乗算合同法を用いてより長い周期の乱数列を生成するためには，64 bit 整数を用いればよい（Fortran では integer *8，C では long long int 宣言を用いる)。

8.2.4 Sequence splitting 法

この方法は，乱数列 $x_0, x_1, \ldots, x_n$ をこの順に $n/p = q$ 個ずつ分割し，各プロセッサに割り当てる方法である（下表参照)。この方法では乗数は a で逐次計算の場合と変らないが，種の作り方が Leapfrog 法と異なってくる。

プロセッサ	乱 数				
P_0	$\boldsymbol{x_0}$	x_1	x_2	$\cdots$	x_{q-1}
P_1	$\boldsymbol{x_q}$	x_{q+1}	x_{q+2}	$\cdots$	x_{2q-1}
P_2	$\boldsymbol{x_{2q}}$	x_{2q+1}	x_{2q+2}	$\cdots$	x_{3q-1}
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
P_i	$\boldsymbol{x_{iq}}$	x_{iq+1}	x_{iq+2}	$\cdots$	$x_{(i+1)q-1}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
P_{p-1}	$\boldsymbol{x_{(p-1)q}}$	$x_{(p-1)q+1}$	$x_{(p-1)q+2}$	$\cdots$	x_{n-1}

課題

1. 付録にある Leapfrog 法の並列プログラムを参考に Sequence splitting 法の並列プログラムを作成し，性能を比較せよ。

2. 64 bit 整数を用いて周期の長い乱数列を生成する並列プログラムを作れ。ただし，$m = 2^e$ とした場合，乗数は $a \equiv 3, 5 \pmod 8$ とし，種 x_0 は奇数とすると，最長周期 2^{e-2} が実現される [9]。

付録 Leapfrog 法の並列プログラム (OpenMP による並列プログラム)

```
 1: ******************************************************
 2: *     Leapfrog method                                *
 3: *     並列に乗算合同法乱数を生成                     *
 4: *     乱数の個数 n, プロセッサ数 p                   *
 5: *                                                    *
 6: *     x_{j+1}=a x_j  (mod m),  m=2^31                *
 7: *                                                    *
 8: ******************************************************
 9:       integer a,aa,seed,n,p,np,ix,ip,j,sum,omp_get_thread_num
10:       parameter (p=4, n=480000000, m1=2147483647)
11:       integer cnt(0:p-1),sd(0:p-1),cntp
12:       real *8 x,z0,z1,pi,true
13:       real *4 t0,t1,tt(2),etime
14: *
15:       t0=etime(tt)
16:       a=65539
17:       seed=133315
18:       write (*,1000) n, p
19:       write (*,1100) a, seed
20: *
21: *    並列計算の前処理部分
22: *
23:       np=n/p
24: *
25: *     各プロセッサの種を計算する
26: *
27:       ix=seed
28:       sd(0)=ix
29:       do ip=1,p-1
30:         call xrand(a,ix,x)
31:         sd(ip)=ix
32:       enddo
33: *
34: *     並列計算用の乗数 a^p (mod m) を計算する
35: *
36:       aa=a
37:       do ip=2,p-1
38:         aa=iand(aa*a,m1)
39:       enddo
40: *
41: *     並列計算開始
42: *
43: *$omp parallel private(ip,z0,z1,ix,j,cntp)
44:       ip=omp_get_thread_num()
45: *
46: *     各 thread でカウンタをリセットし, 種を取得する
47: *
48:       cntp=0
49:       ix=sd(ip)
50: *
51:       call xrand(aa,ix,z0)
52:       do j=1,np-1
53:         call xrand(aa,ix,z1)
54:         if (z0**2+z1**2.le.1.0) then
55:           cntp=cntp+1
56:         endif
57:         z0=z1
58:       enddo
59:       cnt(ip)=cntp
60: *$omp end parallel
61: *
62:       t1=etime(tt)
63: *
64:       do ip=0,p-1
65:         write (*,1200) ip,cnt(ip)
66:       enddo
67: *
```

```
 68:       sum=0
 69:       do ip=0,p-1
 70:         sum=sum+cnt(ip)
 71:       enddo
 72: *
 73:       pi=4.d0*dfloat(sum)/dfloat(n)
 74:       true=4.d0*datan(1.d0)
 75:       write (*,1300) pi,pi-true
 76:       write (*,1400) t1-t0
 77: *
 78:  1000 format (' n=',i10,', p=',i3)
 79:  1100 format (' a=',i10,', seed=',i6)
 80:  1200 format (' cnt(',i1,')=',i8)
 81:  1300 format (' pi=',1pd15.7,', error=',1pe12.3)
 82:  1400 format (' time=',1pe11.3,' sec')
 83:       end
 84:       subroutine xrand(a,r,x)
 85: *
 86: *       r_{j}=a*r_{j-1}  mod m
 87: *       x_j=r_j/m
 88: *
 89:       integer a,r,m1
 90:       real *8 x,xmax
 91:       parameter (m1=2147483647, xmax=2147483647.d0)
 92: *
 93:       r=iand(a*r,m1)
 94:       x=r/xmax
 95: *
 96:       return
 97:       end
```

計算結果

```
% pgf77 -mp test.f
% time a.out
   n= 480000000, p=  4
   a=     65539, seed=133315
   cnt(0)=94249525
   cnt(1)=94239160
   cnt(2)=94246649
   cnt(3)=94249526
   pi=  3.1415405D+00, error=  -5.215D-05
   time=  2.460E+00 sec
   9.791u 0.028s 0:02.45 400.4%    0+0k 0+0io 176pf+0w
```

逐次プログラムによる結果

```
   n= 480000000, p=  4
   a=     65539, seed=133315
   cnt(0)=94249525
   cnt(1)=94239160
   cnt(2)=94246649
   cnt(3)=94249526
   pi=  3.1415405D+00, error=  -5.215D-05
   time=  9.550E+00 sec
   9.541u 0.001s 0:09.54 100.0%    0+0k 0+0io 125pf+0w
```

8.3 準乱数による数値積分

m 重積分

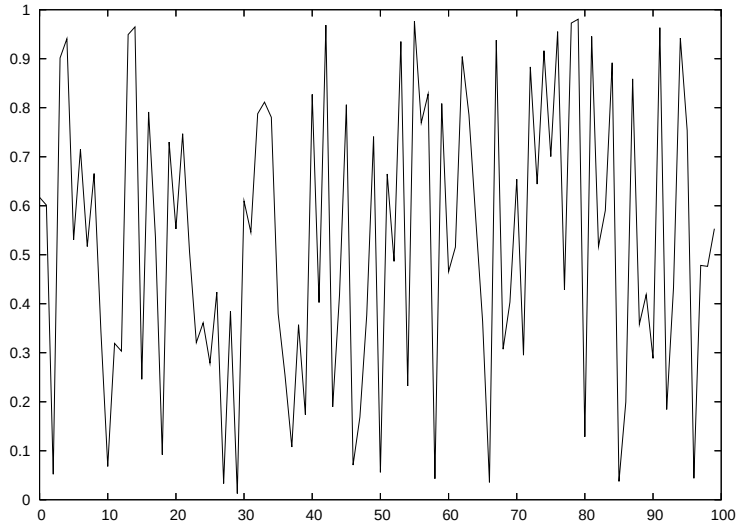
$$I = \int_0^1 \int_0^1 \cdots \int_0^1 f(x_1, x_2, \ldots, x_m)\, \mathrm{d}x_1 \mathrm{d}x_2 \cdots \mathrm{d}x_m \tag{13}$$

を数値的に計算するとき，積分の次元 m がある程度大きくなると，台形公式よりも準乱数を用いた数値積分の方が収束が速くなる。

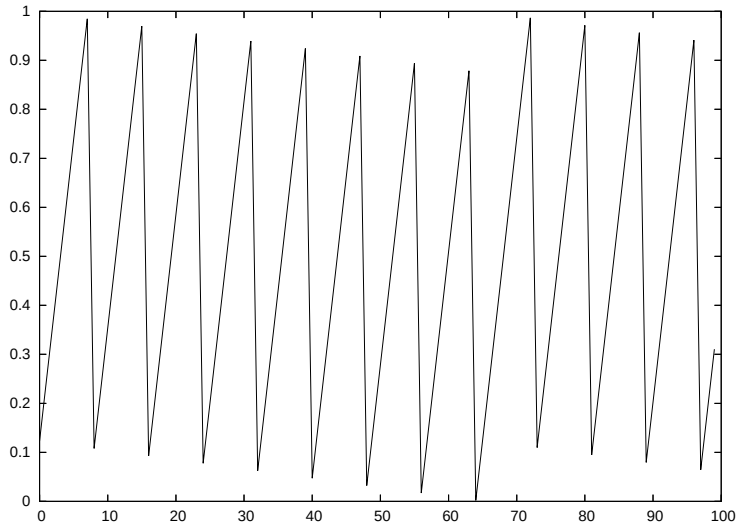
準乱数とは，無理数 α に対して

$$\{n\,\alpha\}, \qquad n = 1, 2, \ldots$$

によって生成された数列のことを言う。ここで $\{x\}$ は x の小数部を表している。準乱数は，疑似乱数に比べると規則性が強い乱数であるが，逆に一様性に優れている乱数であ。ここで時系列として見たときの疑似乱数と準乱数を図に示しておく。



合同法による疑似乱数



準乱数 ($\alpha = \sqrt{17}$ の場合)

一次独立な m 個の二次の無理数 (有理数を係数とする二次方程式の解となる無理数）$\alpha_1, \alpha_2, \ldots, \alpha_m$ を用意し

$$I_N = \frac{1}{N} \sum_{n=1}^{N} f(\{n\alpha_1\}, \{n\alpha_2\}, \ldots, \{n\alpha_m\}) \tag{14}$$

を計算すると，f が連続であれば

$$\lim_{N \to \infty} I_N = I \tag{15}$$

となることが知られている。収束の速さは次元 m に関係なく

$$|I_N - I| = O(N^{-1})$$

である。一方，m 重積分に対する台形公式合は

$$T_N = \frac{1}{N} \sum_{i_1=0}^{n}{}' \sum_{i_2=0}^{n}{}' \cdots \sum_{i_m=0}^{n}{}' f(i_1 h, i_2 h, \ldots, i_m h), \qquad h = 1/n, \quad N = n^m$$

となる。ここで,

$$\sum_{i=0}^{n}{}' a_i = \frac{1}{2}a_0 + a_1 + \cdots + a_{n-1} + \frac{1}{2}a_n,$$

である。収束の速さは

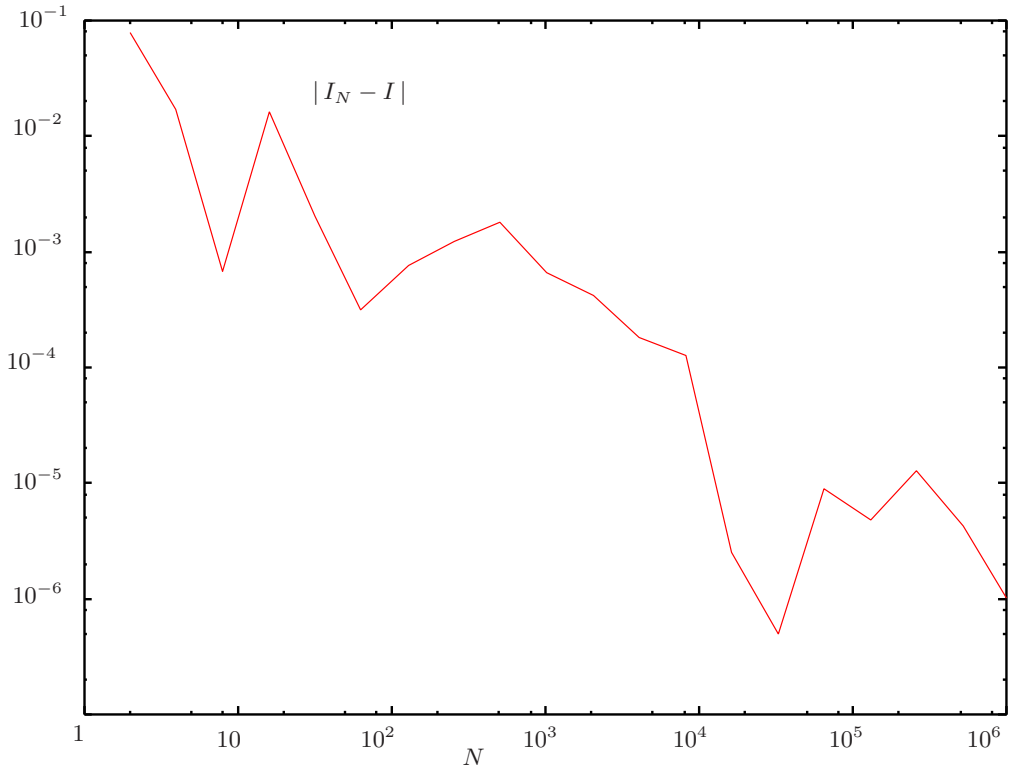
$$|T_N - I| = O(n^{-2}) = O(N^{-2/m})$$

となり，次元に依存する。

数値例 4 重積分

$$I = \int_0^1 \int_0^1 \int_0^1 \int_0^1 \exp(-x_1 - x_2 - x_3 - x_4)\,\mathrm{d}x_1\,\mathrm{d}x_2\,\mathrm{d}x_3\,\mathrm{d}x_4 = (1 - \mathrm{e}^{-1})^4$$

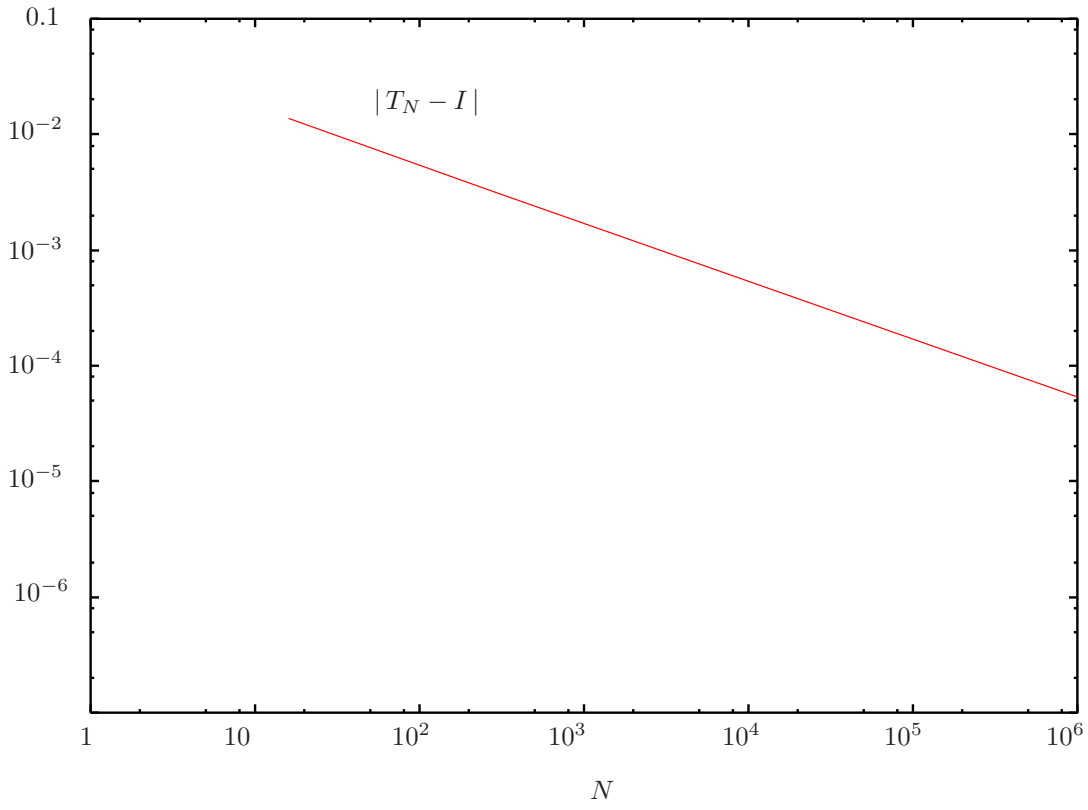
を準乱数を用いて計算する。ここで “種” は $\alpha_1 = \sqrt{11}$, $\alpha_2 = \sqrt{739}$, $\alpha_3 = \sqrt{929}$, $\alpha_4 = \sqrt{15641}$ とする。誤差 $Err = |I - I_N|$ の変化を下に示す。



逐次プログラム

```
 1: Program Quasi_Monte
 2:   implicit none
 3:   integer :: n,NN=10**5
 4:   real (8):: x1,x2,x3,x4,s=0.,frac,f,q,true,err
 5:   real (8):: g1,g2,g3,g4
 6:
 7:   g1=sqrt(11.d0); g2=sqrt(739.d0); g3= sqrt(929.d0); g4=sqrt(15641.d0)
 8:   true=(1.d0-exp(-1.d0))**4   ! 真の値
 9:
10:   do n=1,NN
11:      x1=frac(n*g1); x2=frac(n*g2); x3=frac(n*g3); x4=frac(n*g4)
12:      s=s+F(x1,x2,x3,x4)
13:   end do
14:   q=s/dfloat(NN)
15:   err=abs(q-true)
```

```
16:   write(*,'(i10,1pe12.3)') NN,err
17:
18: end Program Quasi_Monte
19:
20: Function frac(x)   ! 小数部を求める関数
21:   real (8) :: frac,x
22:   frac=x-dint(x);
23:   return
24: end Function frac
25:
26: Function F(x1,x2,x3,x4)   ! 被積分関数
27:   real (8):: x1,x2,x3,x4,F
28:   F=exp(-x1-x2-x3-x4)
29:   return
30: end Function F
```



並列プログラム

```
 1: Program Quasi
 2:   implicit none
 3:   integer :: n,n_max=10**5
 4:   real (8):: x1,x2,x3,x4,s=0.,frac,f,q,true,err
 5:   real (8):: g1,g2,g3,g4
 6:
 7:   g1=sqrt(11.d0); g2=sqrt(739.d0); g3= sqrt(929.d0); g4=sqrt(15641.d0)
 8:   true=(1.d0-exp(-1.d0))**4
 9:
10: !$omp parallel do reduction(+: s) private(x1,x2,x3,x4,x5,x6)
```

```
11:   do n=1,n_max
12:      x1=frac(n*g1); x2=frac(n*g2); x3=frac(n*g3); x4=frac(n*g4);
13:      s=s+F(x1,x2,x3,x4,x5,x6)
14:   end do
15:
16:   q=s/dfloat(n_max)
17:   err=abs(q-true)
18:   write(*,'(i10,1pe12.3,1pe12.3)') n_max,err,t1-t0
19:
20: end Program Quasi
```

6 重積分を $n = 2^{28}$ 点で並列計算した場合の計算時間

p	time [sec]	S_p
1	48.4	1
2	25.1	1.93
3	16.8	2.88
4	13.4	3.61

8.4 共役勾配法の並列化

本節では共役勾配法（Conjugate Gradient method[6, 1, 3] : 以下 CG 法）の並列化について述べる。まず，CG 法について説明する。CG 法は，係数行列 $A \in \mathbb{R}^{n \times n}$ と定数ベクトル $\boldsymbol{b} \in \mathbb{R}^n$ を持つ連立一次方程式

$$A\boldsymbol{x} = \boldsymbol{b} \tag{16}$$

を解くための方法の一つである。ただしここで，A は対称でなければならない。

方程式 (16) は多自由度を持つ方程式の中で最も基礎的なものの一つであり，そのため同様に多自由度を持つ他の種類の方程式（常微分方程式や代数方程式など）の数値計算においてもしばしば方程式 (16) を解く必要が発生する。従って方程式 (16) の解法の求解に必要となる計算時間を並列化等によって短縮することは重要なことであると言える。

方程式 (16) の解法は数多くあるが，その中で CG 法は反復法と呼ばれる部類に属する。反復法とは，一定の計算を繰り返し行なう（反復と呼ぶ）ことで近似解を改良していく方法の総称であり，一般に方程式 (16) の係数行列 A を改変しないため，A が疎（ほとんどの要素が 0）である場合に有用であるとされる。ただし反復法には欠点として，近似解の収束までに必要となる反復回数を事前に予測することが困難であるということがある。その中にあって CG 法は，理論的には “必ず n 回の反復で収束する” という利点を持つことから，その発表当初（1952 年）には大きな話題となった。しかし実際には丸め誤差の影響で n 回では収束せず，特に A の条件数が大きい場合には反復が収束しない場合もあるなど，誤差に対する脆さが判明することでその話題は熱を失なった。しかし CG 法の有用性が否定された訳ではなく，特に方程式 (16) に前処理を適用することで収束性を大きく改善できることもあり，方程式 (16) を解く際に有力な選択肢の一つとなっている。

CG 法のアルゴリズムを以下に示す。

1: $\boldsymbol{x}$ に初期値を設定，$\boldsymbol{r} \leftarrow \boldsymbol{b} - A\boldsymbol{x}$，$\gamma \leftarrow \boldsymbol{r}^T\boldsymbol{r}$，$\beta \leftarrow 0$。
2: **do** until convergence
3: $\quad \boldsymbol{u} \leftarrow \boldsymbol{r} + \beta\boldsymbol{u}$.
4: $\quad \boldsymbol{v} \leftarrow A\boldsymbol{u}$.
5: $\quad \delta \leftarrow \boldsymbol{u}^T\boldsymbol{v}$.
6: $\quad \alpha \leftarrow \gamma/\delta$.
7: $\quad \boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha\boldsymbol{u}$.
8: $\quad \boldsymbol{r} \leftarrow \boldsymbol{r} - \alpha\boldsymbol{v}$.
9: $\quad \tilde{\gamma} \leftarrow \boldsymbol{r}^T\boldsymbol{r}$.
10: $\quad \beta \leftarrow \tilde{\gamma}/\gamma$.
11: $\quad \gamma \leftarrow \tilde{\gamma}$.
12: **end do**.

上に示すように，CG 法の反復計算は，ベクトルの線形結合（3，7，8 行目），ベクトルの内積（5，9 行目），行列とベクトルの積（4 行目），スカラの除算（6，10 行目）によって構成される。このうち，スカラの除算は並列化が困難であるが，n が充分大きい場合はスカラの演算は全体の計算量と比較して無視できる程度となるため，無理に並列化を行なう必要はない。そこで，線形結合，内積，行列・ベクトル積の並列化について，Fortran と MPI または OpenMP（ループ並列化）との組み合わせによる実装方法を以下で説明する。

準備として，ベクトルおよび行列のデータ形式について説明しておく。

ベクトル ベクトル $\boldsymbol{v}$ の各要素は一般に $v_i\ (i = 1, \ldots, n)$ で表わされ，その並びは一次元的である。そのため，プログラム中でベクトルの値を表現する際は一次元配列を用いるのが一般的である。つまり，

$$\texttt{v(i)} = v_i$$

とする。

ここで，ベクトルの線形結合および内積を逐次計算するプログラムを示しておく。

- 線形結合（$\boldsymbol{w} = \boldsymbol{u} + \alpha\boldsymbol{v}$）

```
do i=1,n
   w(i)=u(i)+alpha*v(i)
end do
```

- 内積（$\gamma = \boldsymbol{u}^T\boldsymbol{v}$）

```
gamma=0.d0
do i=1,n
   gamma=gamma+u(i)*v(i)
end do
```

行列 上で述べたベクトルの場合と同様に考えるならば，行列 A の各要素 $a_{i,j}$ $(i,j=1,\ldots,n)$ は二次元的に並んでいるため，プログラム中での表現には二次元配列を用いるのが一般的である。しかしここでは A は疎，つまり $a_{i,j}$ のほとんどが 0 である場合を想定しているため，その全てをデータとして保持するのは甚だ効率が悪い。このような場合は，圧縮行格納法（Compressed Row Storage[1] : 以下 CRS）を用いる。その説明のための例として，A が

$$A=\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 \\ 1 & 3 & 0 & 1 & 0 & 2 \\ 0 & 0 & 2 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 2 & 1 & 0 & 1 & 3 \end{pmatrix} \tag{17}$$

である場合を考える。CRS では 3 つの一次元配列 `val`（実数型），`col_ind`（整数型），`row_ptr`（整数型）を用いて行列を表現し，上の場合ではそれぞれ

val	3	1	1	3	1	2	2	1	1	3	1	1	2	1	1	3
col_ind	1	2	2	1	4	6	3	6	2	4	5	6	2	3	5	6

row_ptr	1	3	7	9	11	13	17

というように格納される。ここで `val` は A の非ゼロ要素を格納するための配列で，行番号の若い順に格納されている。`col_ind` は `val` の対応する位置の要素の列番号を格納する配列である。`row_ptr` は各行の非ゼロ要素の列が `val` の何番目から始まっているかを指し示す配列であり，第 `i` 行の開始位置は `row_ptr(i)` に示されている。なお，`row_ptr(n+1)` は 非ゼロ要素の総数 + 1 が格納されている。

CRS によって行列 A のデータが納まっている場合について，$\boldsymbol{v}=A\boldsymbol{u}$ を逐次計算するプログラムを以下に示す。

```
do i=1,n
   v(i)=0.d0
   do j=row_ptr(i),row_ptr(i+1)-1
      v(i)=v(i)+val(j)*u(col_ind(j))
   end do
end do
```

8.4.1 OpenMP による並列化

線形結合，内積，行列・ベクトル積ともに単純なループで構成されるため，OpenMP のループ並列化機能を用いることで容易に並列化できる。それぞれのプログラムを以下に示す。

線形結合 $\boldsymbol{w}=\boldsymbol{u}+\alpha\boldsymbol{v}$.

```
!$omp parallel do
      do i=1,n
         w(i)=u+alpha*v
      end do
!$omp end parallel do
```

内積 $\gamma=\boldsymbol{u}^T\boldsymbol{v}$.

```
      gamma=0.d0
!$omp parallel do reduction(gamma)
      do i=1,n
```

```
         gamma=gamma+u(i)*v(i)
      end do
!$omp end parallel do
```

行列・ベクトル積 $\boldsymbol{v} = A\boldsymbol{u}$.

```
!$omp parallel do
      do i=1,n
         v(i)=0.d0
         do j=row_ptr(i),row_ptr(i+1)-1
            v(i)=v(i)+val(j)*u(col_ind(j))
         end do
      end do
!$omp end parallel do
```

内積のみ，一つの値に全ての計算が集約されているため，注意が必要となる。

8.4.2 MPI による並列化

P 個のプロセッサによって並列化する場合を考える。MPI によって並列化を行なう場合は，最初にプロセッサによるデータの分割を定義する必要がある。今回は，各ベクトルを分割する形を考え，ベクトル $\boldsymbol{v}$ のうちプロセッサ p $(p = 1, \ldots, P)$ の割り当てを $v_{m_p}, \ldots, v_{n_p}$ とする（ただし $m_1 = 1$, $n_p + 1 = m_{p+1}$）。この場合，内積および線形結合のプログラムは以下のようになる。

線形結合 $\boldsymbol{w} = \boldsymbol{u} + \alpha\boldsymbol{v}$

```
      do i=mp,np
         w(i)=u(i)+alpha*v(i)
      end do
```

内積 $\gamma = \boldsymbol{u}^T\boldsymbol{v}$

```
      gamma_local=0.d0
      do i=mp,np
         gamma_local=gamma_local+u(i)*v(i)
      end do
      call MPI_ALLREDUCE(gamma_local,gamma,1,
     &   MPI_DOUBLE_PRECISION,MPI_SUM,0,comm,ierr)
```

内積の計算は，最後に全てのプロセッサによる総和をとる必要から `MPI_AllReduce` を用いている。

次に行列・ベクトル積 $\boldsymbol{v} = A\boldsymbol{u}$ の並列化について説明する。行列が CRS によって圧縮されていると想定すると，行列の要素は行ごとに納められているため，そのデータをプロセッサ間で分割する場合は行ごとに分割するのが一般的である。つまり，プロセッサ p は行列 A のうち m_p 行目から n_p 行目までを保持する。ここで式 (17) の行列 A とベクトル $\boldsymbol{u}$ の積を，$P = 2$, $m_1 = 1$, $n_1 = 3$, $m_2 = 4$, $n_2 = 6$ で並列化する場合を考える。プロセッサ 1 は A の 1〜3 行目を保持することになるが，2 行目と 3 行目は 4 列目と 6 列目に 0 でない値を持つため，これらの行とベクトルの積には $\boldsymbol{u}$ の要素のうち u_4 と u_6 が必要となる。しかしこれらの値はプロセッサ 2 が保持しているので，行列・ベクトル積の計算前にこれらの値をプロセッサ 2 からプロセッサ 1 へ渡す必要がある。同様に，4 行目と 6 行目は 2 列目と 3 列目に 0 でない値を持つため，積の計算前にプロセッサ 1 からプロセッサ 2 へ u_2 と u_3 を渡す必要がある。ただし通信コスト抑制のために，互いに全ての要素を送受信するのではなく，積の計算に必要な要素のみを送受信の対象とし，必要ない要素は送受信しないように注意する必要がある。

以上の事柄を一般化するため，表 3 に示す配列を用いる。また，ベクトル $\boldsymbol{u}$ を納める配列 u は，

u_{m_p}	...	u_{n_p}	（他のプロセッサから受信した要素）

のように用いる。CRS で圧縮された A のデータもこれに合わせる必要がある。例えば，式 (17) を上と同様に分割する場合は，各配列を表 4 のように設定する。

以上の配列を用いて，$\boldsymbol{v} = A\boldsymbol{u}$ を MPI によって並列化したプログラムを以下に示す。

表 3: MPI による行列・ベクトル積の並列化で必要となる配列

配列名	意味
send_ind(:,q)	プロセッサ q へ送信する要素の添字の列
recv_ptr(q)	プロセッサ q から受信した値を配列に納める位置
nsend(q)	プロセッサ q へ送信する要素の数
nrecv(q)	プロセッサ q から受信する要素の数

表 4: 式 (17) の A による $\boldsymbol{v} = A\boldsymbol{u}$ を MPI で並列化する際の配列の値

	プロセッサ 1								プロセッサ 2							
u	u_1	u_2	u_3	u_4	u_6				u_4	u_5	u_6	u_2	u_3			
val	3	1	1	3	1	2	2	1	1	3	1	1	2	1	1	3
col_ind	1	2	1	2	4	5	3	5	4	1	2	3	4	5	2	3
row_ptr	1	3	7	9					1	3	5	9				
send_ind(:,q)	2	3	($q=2$)						1	3	($q=1$)					
recv_ptr	1	4							4	1						
nsend	0	2							2	0						
nrecv	0	2							2	0						

```
 do q=1,P
    if(p.ne.q) then
       do j=1,nsend(q)
          sendbuf(j,q)=u(send_ind(j,q))
       end do
       call MPI_ISEND(sendbuf(:,q),nsend(q),
&         MPI_DOUBLE_PRECISION,q,tag,ierr,sendreq(q))
       call MPI_IRECV(u(recv_ptr(q)),nrecv(q),
&         MPI_DOUBLE_PRECISION,q,tag,ierr,recvreq(q))
    end if
 end do
 do q=1,Np
    if(p.ne.q) MPI_WAIT(recvreq(q))
 end do
 do i=mp,lp
    v(i)=0.d0
    do j=row_ptr(i),row_ptr(i+1)-1
       v(i)=v(i)+val(j)*u(col_ind(j))
    end do
 end do
```

上のプログラムでは，事前の通信は送信／受信が集中するため，デッドロックに陥いらないようにノンブロッキング通信を用いている。また，通信に必要となる tag の計算は省略しているが，実際には送信側と受信側の組み合わせによって固有の値となるように注意して設定する必要がある。

また上でも述べたように，通信コスト抑制の必要から，ベクトル要素の送受信は全ての要素が対象になる訳ではなく，必要な要素のみを対象とする。一般的に，行列 A が疎で，その構造が規則的であれば，通信コストをより抑えやすい。例として，ある二次元領域内で偏微分方程式を差分法などで離散化し，得られる連立一次方程式の解の計算を並列化する場合を考える（図 5)。問題の規模 n は領域の面積に比例するが，送受信の対象となるデータは領域をプロセッサごとの小領域に分割することで発生する境界の長さに比例する。つまり通信量は $O(\sqrt{n})$（より正確には $O(\sqrt{nP})$）に比例すると考えられる。

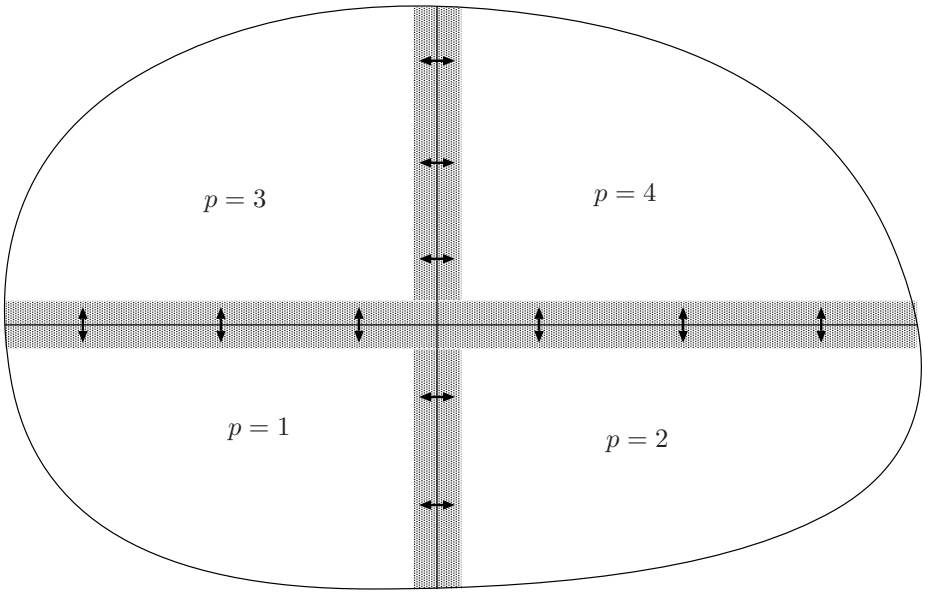


図 5: 二次元領域の分割：網掛けされた部分のデータが送受信の対象となる

9 付録

9.1 MPICH のインストール

MPICH はアルゴンヌ国立研究所 (Argonne National Laboratory) のホームページ [13] から最新版をダウンロードすることができる。2008 年 3 月 30 日現在，MPICH の最新版のバージョンは 1.2.7p1 であるので，`mpich-1.2.7p1.tar.gz` をダウンロードする。

各マシンには Linux がインストールされているとして，これらがイーサネットで接続されているとする。またユーザのホームディレクトリは NFS により共有されているとする。

```
# tar zxvf mpich-1.2.7p1.tar.gz
# cd mpich-1.2.7p1
# ./configure --prefix=/usr/local/mpich-1.2.7p1
# make
# make install
```

インストール後の設定として，まず machines ファイル (`/usr/local/mpich-1.2.7p1/share/machines.LINUX`) に並列計算させる計算機名を書き込む。

machines.LINUX

```
host1
host2
...
```

最後に環境変数 `PATH` に`/usr/local/mpich-1.2.7p1/bin` を加えれば，MPICH が使用可能になる。

9.2 Neumann のアーキテクチャ

参考文献

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. van der Vorst, Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods, SIAM, Philadelphia, 1993.

[2] Rohit Chandra et al. “Parallel Programming in OpenMP,” Morgan Kaufmann Publ., 2001.

[3] J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. van der Vorst, Numerical Linear Algebra for High-Performance Computers, SIAM, Philadelphia, 1998.

[4] “地球シミュレータ”，応用数理 Vol.16, No. 2 (Jun 2006), pp.2–35.

[5] “地球シミュレータ（続）”，応用数理 Vol.16, No 3, (Sep 2006), pp.2-19.

[6] 藤野清次，張紹良，反復法の数理，朝倉書店，東京，1996.

[7] `http://accc.riken.jp/HPC/HimenoBMT/`

[8] 岩崎洋一，“次世代スーパーコンピュータと計算機科学の発展”，応用数理 Vol.16, No. 2 (Mar 2007), pp.57–64.

[9] D.E. Knuth, “The Art of Computer Programming (Vol. 2, 3rd Ed.),” Addison–Wesley, 1998.

[10] `http://www.netlib.org/linpack/`

[11] Michael J. Quinn, Parallel Programming in C with MPI and OpenMP, International Edition, 2003.

[12] J.J. Modi, “Parallel Algorithms and Matrix Computation,” Oxford University Press, 1988.

[13] http://www-unix.mcs.anl.gov/mpi/mpich1/

[14] http://www.mpi-forum.org/

[15] `http://www.top500.org/`

[16] P. パチェコ，MPI 並列プログラミング, 培風館 (2001).

[17] http://www.csm.ornl.gov/pvm/

[18] Michael J. Quinn, “Parallel Programming in C with MPI and OpenMP,” McGraw–Hill, 2004.

[19] `http://www.spec.org/`