コンピュータシステム　(2)

坂井　修一
東京大学大学院 情報理工学系研究科 電子情報学専攻

- はじめに
- 並列計算機のプログラミング
- プログラマの３つの態度
- 例題による検討
- （ICTは何のために？）
- SIMD/MIMD
- メッセージ交換と共有メモリ
- 並列計算機の分類

はじめに

- 講義内容：　コンピュータシステム
  - 並列処理、再構成、OS、セキュリティ、スーパコンピュータ、コンピュータの未来
- 関連講義（電子・情報系）：
  - 論理回路基礎：入江、２年冬
  - コンピュータアーキテクチャ：坂井、３年夏
  - アドバンスト・コンピュータアーキテクチャ：入江、大学院奇数年度夏
- 成績
  - 出席＋レポート

コンピュータシステム　予定

第1回　10月　1日（月）　イントロ、東日本大震災とICT
第2回　10月15日（月）　並列計算機概論（1）
　　10月22日（月）　休講
第3回　10月 29日（月）　並列計算機概論（2）
第4回　11月　5日（月）　並列計算機概論（3）
第5回　11月　8日（水）　3D LSIと動的再構成アーキテクチャ
第6回　11月12日（月）　マイクロプロセッサ・アーキテクチャ(1)
第7回　11月 19日（月）　超スマート社会のエッジコンピューティング
第8回　11月 26日（月）　マイクロプロセッサ・アーキテクチャ(2)
第9回　12月　3日（月）　スーパコンピュータ
第10回　12月17日（月）　ITと行政
第11回　12月26日（月）　最新オペレーティングシステム

並列計算機のプログラム

- プログラマの態度
  - 従来型の手続き型言語による直列プログラム以外は書く気がない
    - C(++)、FORTRAN、PASCAL
    - コンパイラによる自動並列化
  - 多少の手直しはしてもいい
    - fork/join, DOALL/DOACROSS
    - メッセージ通信： send/receive
    - 同期: sync, barrier
    - データ分配
  - 全く違うコンセプトでプログラムする
    - 関数型言語： Lisp, ML, Ph, Id
    - 論理型言語： KL/1, FLENG
      - 自然に記述すれば自然に並列化される

例. Wavefront問題

$X_{i,j} = X_{i-1,j} + X_{i,j-1}$　　$(i = 0, 1, \ldots N-1,\ j = 0, 1, \ldots N-1)$

- C プログラム（直列型）

```
intitialize();
for  (i=1; i<N; i++)
    for (j=1; j<N; j++)
        X[i,j] = X[i-1][j] + X[i][j-1];
```

- コンパイル後

```
S1: load R1 X[Ri-1][Rj]
    load R2 X[Ri][Rj-1]
    add R3 R1 R2
    add Rj Rj 1
    store R3 X[I][j]
    blt Rj N S1
    add Ri Ri 1
    blt Ri N S1
```

1	1	1	1	1	1	1	1	1	1
1	2	3	4						
1	3	6							
1	4								
1									
1									
1									
1									
1									
1									

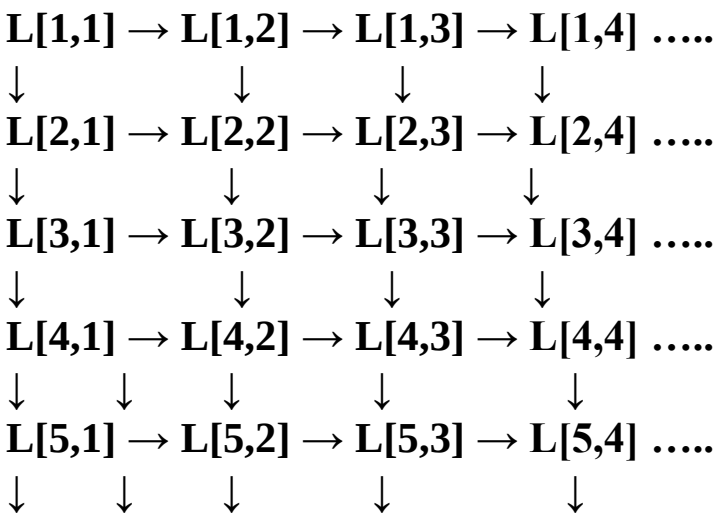
ループ内の命令レベル最適化：
ループアンローリング、
ソフトウェアパイプライニング
ピープホール最適化

→　スーパスカラ処理の最適化（入江）

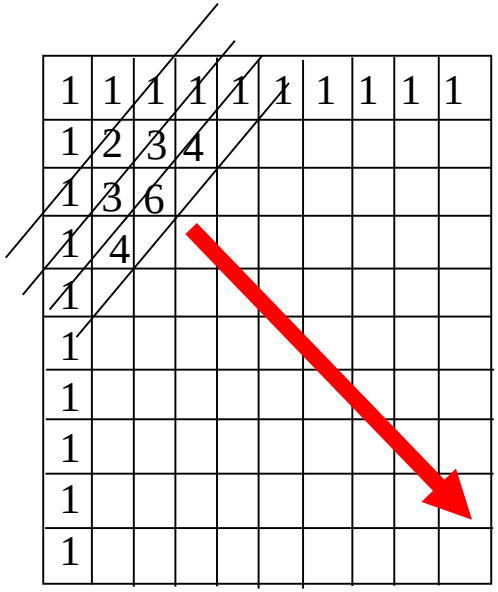
Wavefront問題：ループ間並列処理

$$X_{i,j} = X_{i-1,j} + X_{i,j-1} \qquad (i = 0, 1, \ldots N-1,\ j = 0, 1, \ldots N-1)$$

ループ間の依存性



L[i,j]: Xi,jを求めるループ本体
A→B ： BがAに依存することを表す



依存関係にあるループは直列にしか実行されない

→ 理想並列実行は、左上から右下へ進行する

Wavefront問題の並列化(1): 自動並列化

$X_{i,j} = X_{i-1,j} + X_{i,j-1}$ $(i = 0, 1, \ldots N-1,\ j = 0, 1, \ldots N-1)$

- C プログラム（直列型）

```
intitialize();
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
        X[i][j] = X[i-1][j] + X[i][j-1];
```

1	1	1	1	1	1	1	1	1	1
1	2	3	4						
1	3	6							
1	4								
1									
1									
1									
1									
1									
1									

- コンパイラが自動で並列化する
  - 依存性解析
    - データ依存（フロー依存）
    - 逆依存
    - 制御依存
  - 並列化コードの生成

→ **Wavefrontの例でもわかるように、一般に自動並列化は難しい**

Wavefront問題の並列化(2):並列化プリミティブの記述

$X_{i,j} = X_{i-1,j} + X_{i,j-1}$　　$(i = 0, 1, \ldots N-1,\ j = 0, 1, \ldots N-1)$

- C プログラム（並列型）

```
intitialize();
barrier();
for (m=2; m<N; m++){  /* 左上三角   */
    forall (i=1; i<m; i++){
        j = m − i;
        X[i][j] = X[i−1][j] + X[i][j−1];}
    barrier(); }
for (m=N; m≦2N−2; m++){/* 右下三角   */
    forall (i=m−N+1; i<N; i++){
        j = m − i;
        X[i,j] = X[i−1][j] + X[i][j−1];}
    barrier(); }
```

1	1	1	1	1	1	1	1	1	1
1	2	3	4						
1	3	6							
1	4								
1									
1									
1									
1									
1									
1									

並列化プリミティブ導入の＋／－

- 利点
  - 既存のプログラムからの差異が小さい
  - コンパイラの負担があまりない
  - （特に定型的な問題で）並列性が十分に取り出せる
    - ｛（既存のＣＰＵ）＊Ｎ｝で期待した（程度の）性能を引き出すことができる
  - 並列実行手順がよくわかる
    - デバグや最適化が（慣れれば）やりやすい
- 問題点
  - プログラマの負担増
  - 最大限の並列性を引き出せるとは限らない

宣言型言語の導入

- 宣言型言語： Val, Id, Ph, ML, KL/1, FLENG..
  - 問題の仕様記述がそのままプログラムになる
  - 直列実行を仮定しない
  - 並列性を陽に記述しない (implicit parallelism)
    - 最大限の並列性が自然に抽出される
  - 単一代入規則による依存関係の単純化
    - すべての変数が一回しかWRITEされないため、
      データ依存以外の依存関係が発生しない
  - 関数性： 局所的な引数だけで結果が一意に決まる
    - 「引数→関数値」の流れだけを追えばよい
  - その他の特徴
    - 部分実行、lazy evaluation, polymorphism,etc （授業ではやらない）

Wavefront問題の並列処理(3):宣言型言語による記述

$X_{i,j} = X_{i-1,j} + X_{i,j-1}$ $(i = 0, 1, \ldots N-1,\ j = 0, 1, \ldots N-1)$

- Id プログラム (1)

```
X = make_matrix ((0,N-1), (0, N-1)) (f X)
def f  X(i,j) = if i == 0 then 1
                elseif j == 0 then 1
                  else X[i-1, j]+X[i,j-1]
```

1	1	1	1	1	1	1	1	1	1
1	2	3	4						
1	3	6							
1	4								
1									
1									
1									
1									
1									
1									

- Idプログラム(2)

```
X = {matrix ((0,N-1),(0,N-1)) of
    | [0, 0]  = 1
    | [i,  0]  = 1   || i ← 1 to N-1
    | [0,  j]  = 1   || j ← 1 to N-1
    | [i,   j]  = X[i-1, j]+X[i,j-1] || i ← 1 to N-1;  j ← 1 to N-1}
```

宣言型言語の＋／－

- **利点**
  - 最大限の並列性が自然に取り出せる
    - **並列化コンパイラの作成が簡単**
  - プログラムが書きやすい／読みやすい
    - **問題をありのままに記述すればよい**
  - 実行手順はコンピュータに任せればよい
- 問題点
  - 既存のプログラムからの差が大きい
    - **「すべては一瞬で流れ去る」世界**
  - 並列実行手順がプログラマから見えない
  - 資源を陽に管理・制御するのが難しい
  - 実行時に細かな同期（**Write/Readの順序制御**）が必要

例. 行列のかけ算

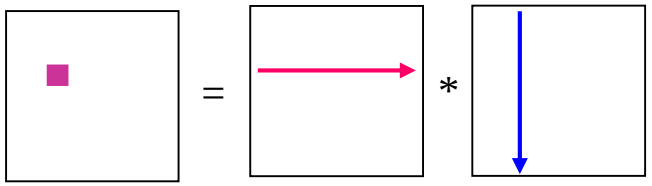
$C_{i,j} = \Sigma_k A_{i,k} * B_{k,j}$　　$(i = 0, 1, \ldots N-1,\ j = 0, 1, \ldots N-1)$

- C プログラム（直列型）

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

並列化できる場所

- iについての繰り返し
- jについての繰り返し
- A[i,k] * B[k, j]　in k loop



行列積の並列化(1)

- コンパイラによる自動並列化
  - iの繰り返し, jの繰り返しは簡単： ループ間に依存関係なし
  - A[i][k] * B[k][j]を切り出せれば、並列化可能

- 並列化プリミティブの導入

```
forall (i=0; i<N; i++)
    forall (j=0; j<N; j++){
        forall (k=0; k<N; k++) D [i][k][j] = A[i][k] * B[k][j];
        for (k=0; k<N; k++) C[i][j] = C[i][j]  + D [i][k][j]; }
```

  - プログラマ、コンパイラともに負担は小さい
  - 並列性が十分に取り出せる
  - 並列実行手順がよくわかる

→ 行列積程度であれば、既存言語によるプログラムの自動並列化やプリミティブの導入程度でよいだろう

行列積の並列化(2)

- 宣言型言語による行列積

```
C = {matrix ((0,N-1),(0,N-1)) of
     | [i,j] = C[i,j] + A[i,k] * B[k,j] || i <- 0 to N-1; j <- 0 to N-1; k <- 0 to N-1; }
```

注. このプログラムは関数的ではない

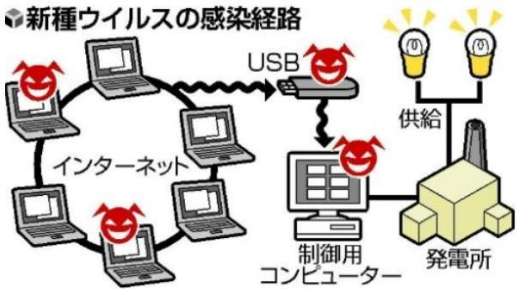
→ 直列プログラムと大差ないが、並列性の意識はいらない

プログラマの３つの態度

	直列＋自動並列化	並列化プリミティブ	宣言型言語
プログラマの負担	なし	大	小
既存プログラムとの差	なし	小	大
コンパイラの負担	大	小	小
並列性の抽出	困難	ある程度	最大限
実行トレース・デバグ 陽な資源管理	難	楽	難

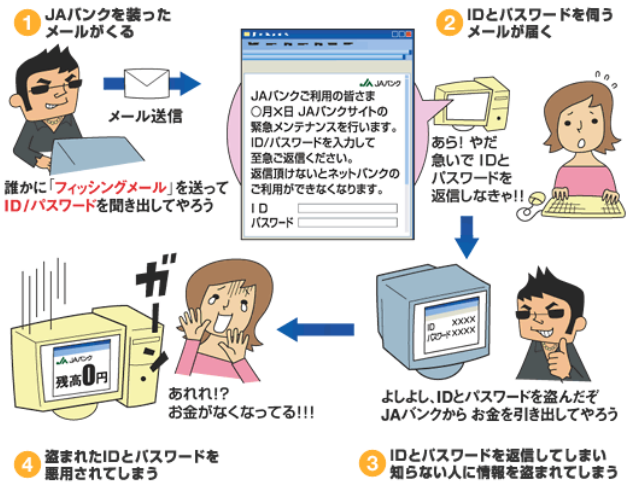
- 補足
  - 並列性の陽な記述：　Data Parallel, Fork/Join, Directive/Pragmaなど
  - 通信ライブラリ：　MPIなど

閑話： ICTは何のために？

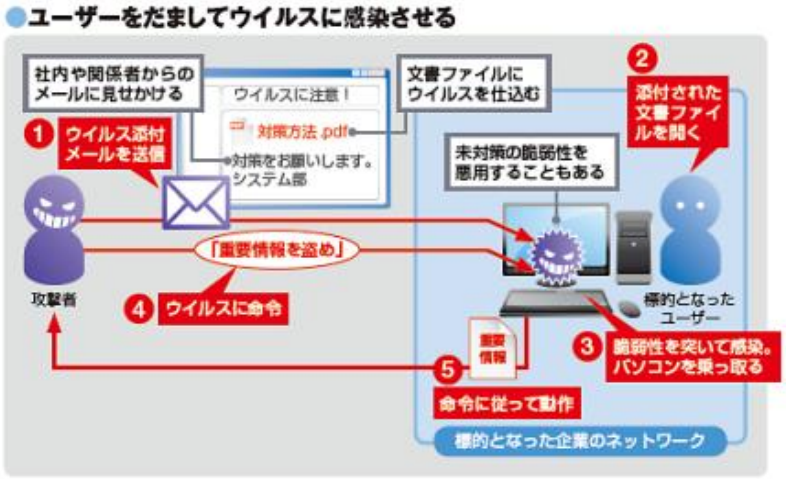


スタックスネットはネットから切り離されたシステムにも感染する(読売新聞、2010年10月4日)

サイバーテロ(Stuxnet)



フィッシング詐欺

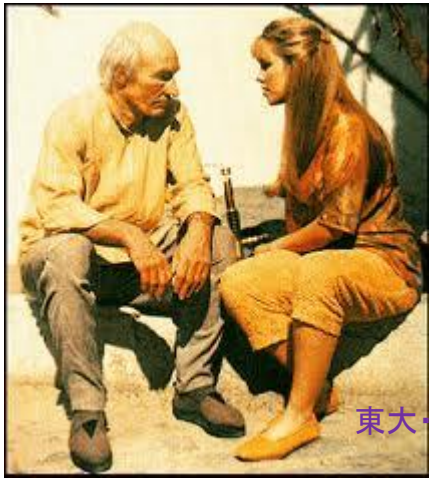


標的型攻撃

遠隔操作ウィルス

ICTは何をめざせばよいか？

- e.g.「スタートレック」の世界
  - 貨幣経済　無し
  - 生存競争　無し
  - 人々がもっぱら人格の陶冶のために人生を送る
- なぜできるのか？
  - ワープ航法：　宇宙資源の活用
  - E ＝ MC^2 ：　エネルギーと物質の変換
    - 転送装置：　瞬間移動
    - レプリケータ：　料理、宇宙船の材料などを作る
  - ホロデッキ：　完璧な仮想現実



「行列のかけ算」:隘路はどこに？

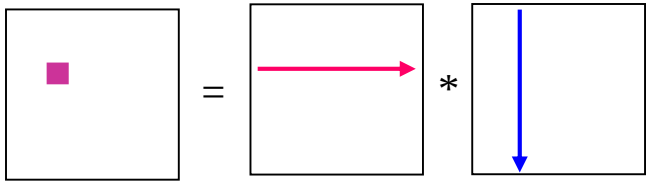
$C_{i,j} = \Sigma_k A_{i,k} * B_{k,j}$　　$(i = 0, 1, \ldots N-1,\ j = 0, 1, \ldots N-1)$

- C プログラム（直列型）

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

並列化できる場所

- iについての繰り返し
- jについての繰り返し
- A[i,k] * B[k, j]　in k loop



- 計算量$O(N^3) \rightarrow O(\log_2 N)$：　ALU/FPUの利用だけを考えた値

- 疑問：A[i][k]、B[k][j]はいつ何度読み出されるか？

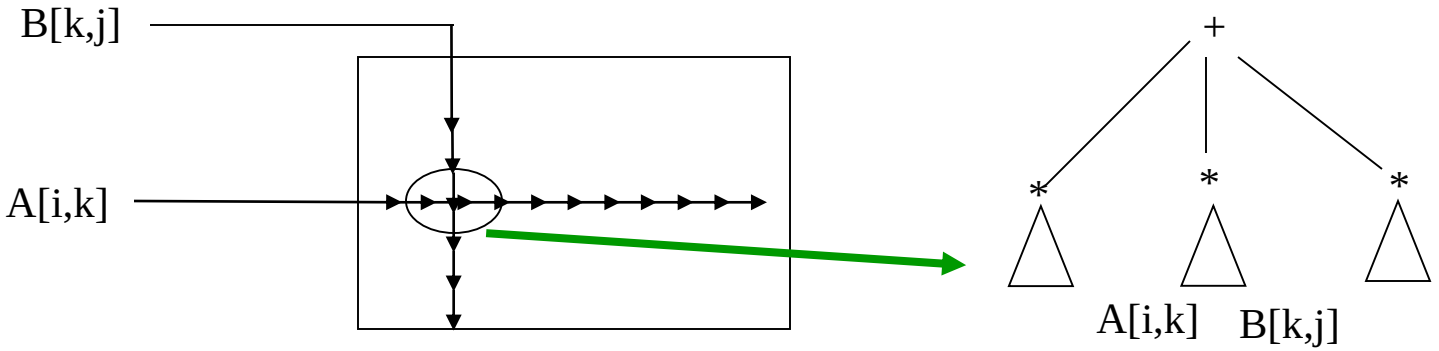
計算とデータ供給

- A[i][k]：

  i −loopで1度。j−loopでN度。k−loopで1度。合計N度読み出し

- B[k][j]：

  i −loopでN度。j−loopで1度。k−loopで1度。合計N度読み出し

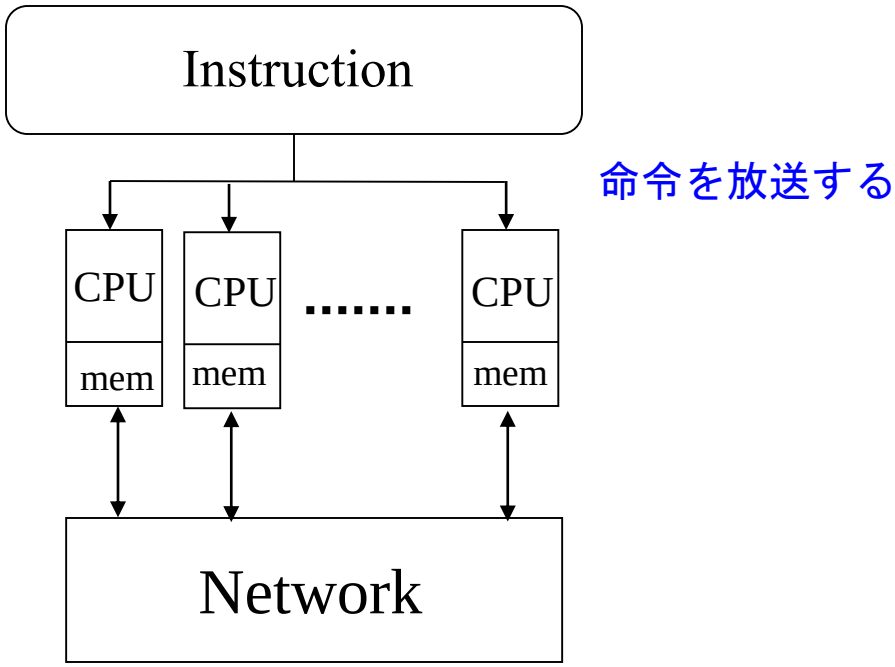


これらのデータが一度に一つずつしか読み出せなかったとすると、最低でもO(N)の実行時間がかかることになる

→　プロセッサではなく、プロセッサにデータを供給するもの（メモリ、バス、ネットワークなど）が隘路となっている

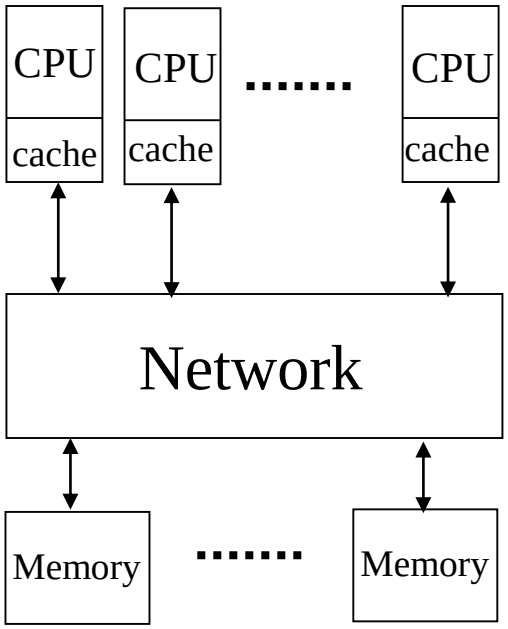
SIMD型並列計算機

- SIMD: Single Instruction Stream Multiple Data Stream
  - 全プロセッサが同一の命令で動作。Data Parallelに近い
  - 命令ごとに動作開始時刻・動作終了時刻が揃えられている（命令単位の同期）
  - 数値計算（例．行列積）など定型問題に有利。複雑な問題には向かない

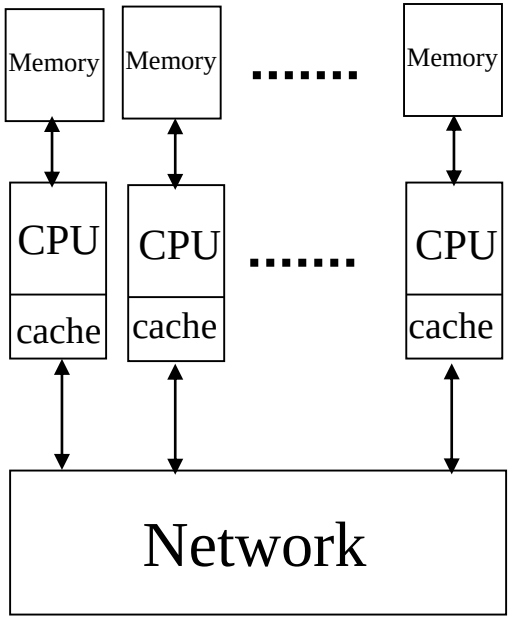


MIMD型並列計算機のアーキテクチャ

- MIMD: Multiple Instruction Stream Multiple Data Stream
- プロセッサ、メモリ、ネットワークの複合体
  - 「ALUの集まったもの」と見るよりも「多数のALUにデータを供給するもの」と見たほうがいい
- 構成方式の基本型は2種類



大域メモリ型



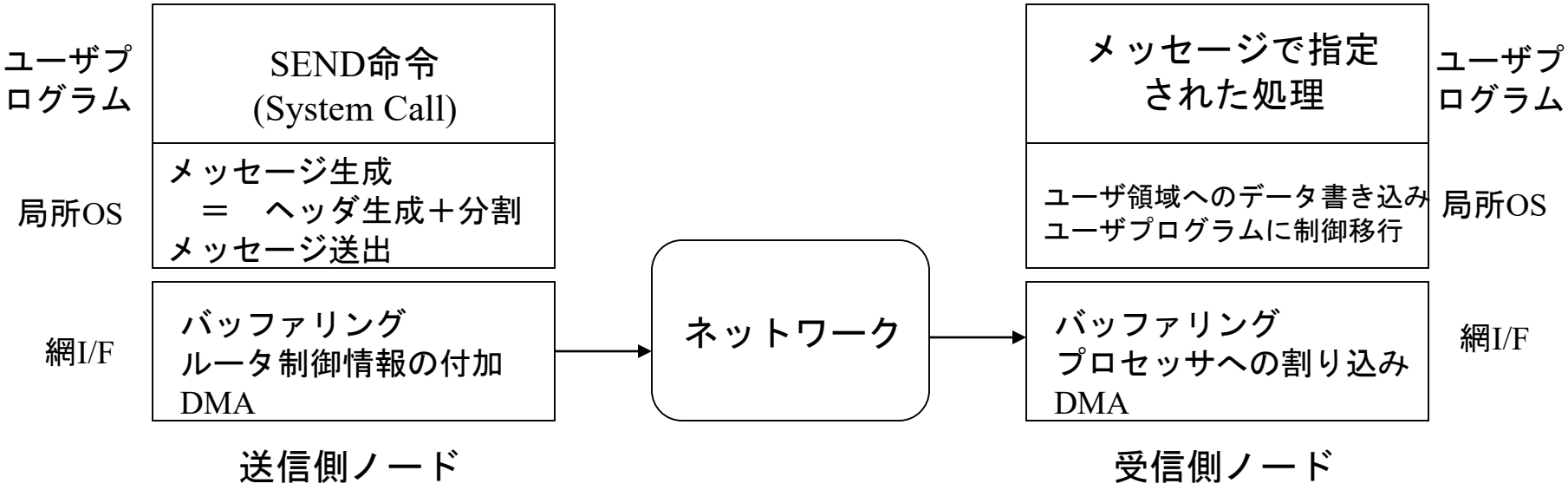
局所メモリ型

メッセージ交換型と共有メモリ型

- 定義
  - メッセージ交換型
    - 各プロセッサ上で走るプロセスがメッセージを送受信することでデータ交換と同期を行う並列処理計算機
  - 共有メモリ型
    - 各プロセッサで走るプロセスが互いに共有メモリを読み書きすることでデータ交換と同期を行う並列処理計算機
- 疑問
  - 上は、構成方式・機構の定義か、それともプログラミングモデルの定義か
- 回答
  - ？？？？（これから検討）

メッセージ交換型並列計算機

- 構成方式
  - 中大規模システム(16プロセッサ以上)が中心
  - 局所メモリ型
- 機構
  - 通常の直列処理＋メッセージ交換による通信・同期



メッセージ交換の方式 [Culler and Singh]

- 同期型(現在型)
  - 送り手: 受け手からのackを待ってユーザプロセスを再開する
  - 受け手： OSがユーザプロセスにデータをコピーし終わったときに、送り手にackを返し、ユーザプロセスを再開する
- (非同期)ブロッキング
  - 送り手:メッセージがOSに渡ると同時にユーザプロセスを再開する
  - 受け手:受け手のOSがユーザプロセスにデータをコピーし終わったとき、ユーザプロセスを再開する(送り手にackを返さない)。
- (非同期)ノンブロッキング
  - 送り手:SEND発行と同時に処理を再開する
  - 受け手:受け手のOSが受信することを決めた時点でユーザプロセスを再開する
  - SEND/RECEIVEとは別に、PROBEが必要

メッセージ交換型の基本操作

Name and Syntax	Function
CREATE(procedure)	Create process that starts at procedure
SEND(src_adr, size, dest, tag)	Send size bytes starting at src_addr to the dest process, with tag identifier
RECEIVE(buffer_addr, size, src, tag)	Receive a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr
SEND_PROBE(tag, dest)	Check if message with identifier tag has been sent to process dest (only for asynchronous non-blocking message passing)
RECV_PROBE(tag, src)	Check if message with identifier tag has been received from process src (only for asynchronous non-blocking message passing)
BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived
WAIT_FOR_END(number)	Wait for number processes to terminate

共有メモリ型

- 構成方式
  - 小規模： 大域メモリ型
  - 大規模： 階層型： 下位は大域メモリ型、上位は局所メモリ型
- 機構
  - メモリ読み書き： 局所メモリも大域メモリも遠隔メモリも一様にread/writeする
  - コピー機構、キャッシュ
  - 同期機構
    - 不可分命令
    - メモリロック
    - バリア同期

※ 詳細は次回

メモリ共有型の基本操作

Name and Syntax	Function
CREATE(p, proc, args)	Create p processes that start executing at procedure proc with arguments args
G_MALLOC(size)	Allocate shared data of size bytes
LOCK(name)	Acquire mutually exclusive access
UNLOCK(name)	Release mutually exclusive access
BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived
WAIT_FOR_END(number)	Wait for number processes to terminate
While (!flag); or WAIT(flag)	Wait for flag to be set (spin or block); used for point-to-point event synchronization
flag = 1; or SIGNAL(flag)	Set flag: wakes up process that is spinning or blocked on flag, if any

プログラム例. Equation Solver

- 直列プログラム

```
int n:
float **A, diff = 0;

main(){
 read(n);
 A <- malloc(a, 2-d array of size n+2
                by n+2 doubles);
 initialize(A);
 Solve(A);
}
```

```
Solve(A)
float **A;
{int i, j, done = 0;
 float diff = 0, temp;
 while(!done) {
   diff = 0;
   for (i=1;i<=n;i++){
     for(j=1;j<=n;j++){
         temp = A[i][j];
         A[i][j]=0.2*(A[i][j]+A[i][j-1]
          +A[i-1][j]+A[i][j+1]+A[i+1][j]);
         diff += abs(A[i][j]-temp);
     }
   }
   if (diff/(n*n)) done = 1;
 }
}
```

並列プログラム(メッセージ交換型)(1/2)

```
int pid,n,nprocs;
float **myA;
main(){
  read(n); read(nprocs);
  CREATE(nprocs-1, Solve);
  Solve();
  WAIT_FOR_END(nprocs-1);
}
```

```
Solve(){
 int i, j, pid, n'=n/nprocs, done=0;
 float temp, tempdiff, mydiff=0;
 myA <- malloc(a 2-d array of size
            [n/procs+2] by n+2);
 initialize(myA);

 while (!done) {
    mydiff = 0;
if (pid !=0)
   SEND(&myA[1,0], n*sizeof(float), pid-1, ROW);
 if (pid = nprocs-1)
   SEND(&myA[n',0], n*sizeof(float), pid+1, ROW);
 if (pid != 0)
   RECEIVE(&myA[0,0], n*sizeof(float), pid-1, ROW);
 if (pid != nprocs-1)
   RECEIVE(&myA[n'+1,0], n*sizeof(float), pid+1,
     ROW);
```

並列プログラム(メッセージ交換型)(2/2)

```
for (i=1; i<=n':i++)
    for (j=1;j<=n;j++){
     myA[i][j]=0.2*(myA[i][j]+myA[i][j-1]
      +myA[i-1][j]+myA[i][j+1]+myA[i+1][j]);
           mydiff += abs(myA[i][j]-temp);
    }
 if (pid != 0) {
   SEND(mydiff, sizeof(float), 0, DIFF);
   RECEIVE(done, sizeof(int),0,DONE):
 }
 else {
  for (i=1:i<=nprocs-1;i++){
    RECEIVE(tempdiff,sizeof(float),*,DIFF);
    mydiff += tempdiff;
  }
if (mydiff/(n*n) < TOL) done = 1;
  for (i=1;i<=nprocs-1;i++)
     SEND(done, sizeof(int),i, DONE);
  }
 }
}
```

→ **通信・同期の操作が多く記述も面倒**

並列プログラム(共有メモリ型)

```
int n,nprocs;
float **A, diff;

LOCKDEC(diff_lock);
BARDEC(bar1);

main(){
  read(n); read(nprocs);
  A <- G_MALLOC(a 2-D array of
            size n+2 by n+2 doubles);
  initialize(A);
  CREATE(nprocs-1, Solve, A);
  Solve();
  WAIT_FOR_END(nprocs-1);
}
```

```
Solve(){
 int i, j, pid, done=0;
 float temp, mydiff=0;
 int mymin = 1+(pid*n/nprocs);
 int mymax = mymin + n/nprocs-1;
while (!done) {
    mydiff = diff = 0;
    BARRIER(bar1, nprocs);
    for (i=mymin; i<=mymax':i++)
   for (j=1;j<=n;j++){
     temp =  A[i][j];
     A[i][j]=0.2*(A[i][j]+A[i][j-1]+A[i-1][j]+ A[i][j+1]
                  +A[i+1][j]);
          mydiff += abs(A[i][j]-temp);}
   LOCK(diff_lock);
   diff += mydiff;
   UNLOCK(diff_lock);
   BARRIER(bar1, nprocs)
 if (diff/(n*n) < TOL) done = 1;
    BARRIER(bar1, nprocs);
 }
}
```

互換性

- メッセージ交換による共有メモリの実現
  - 次のようなメッセージを送信すればよい

(ia, ma, r/w, data, ra)

ia: メモリ操作の入った命令番地
ma: メモリ番地
r/w: 読み出しか書き込みかの区別
data: 書き込みデータ
ra: 戻り番地

  - 受け手では、本メッセージによってメモリ操作を行う
- 共有メモリによるメッセージ交換の実現
  - メモリの特定番地への書き込み・読み出し
  - 同期操作（不可分命令：test&set）が必要

メッセージ交換型と共有メモリ型

- メッセージ交換型と共有メモリ型
  - プログラミングモデルの定義である！

  （歴史的にはそうではなかった）
- 並列計算機における抽象レベル

抽象レベル	具体的な中身
応用	数値計算、AI、DB
プログラミングモデル	メッセージ交換型、共有メモリ型
（抽象化された）通信機構	メッセージ生成・送受信、同期方式
ハードウェア	プロセッサ、ネットワーク

メッセージ交換型と共有メモリ型の○×

	メッセージ交換型	共有メモリ型
プログラミングの容易さ	×	○
効率良く実現するためのハードウェアコスト	小	大
プログラムの正当性の証明	○ （ブロッキング型）	△
コンパイラによる並列化	△	○
分散OSの作りやすさ	△	○

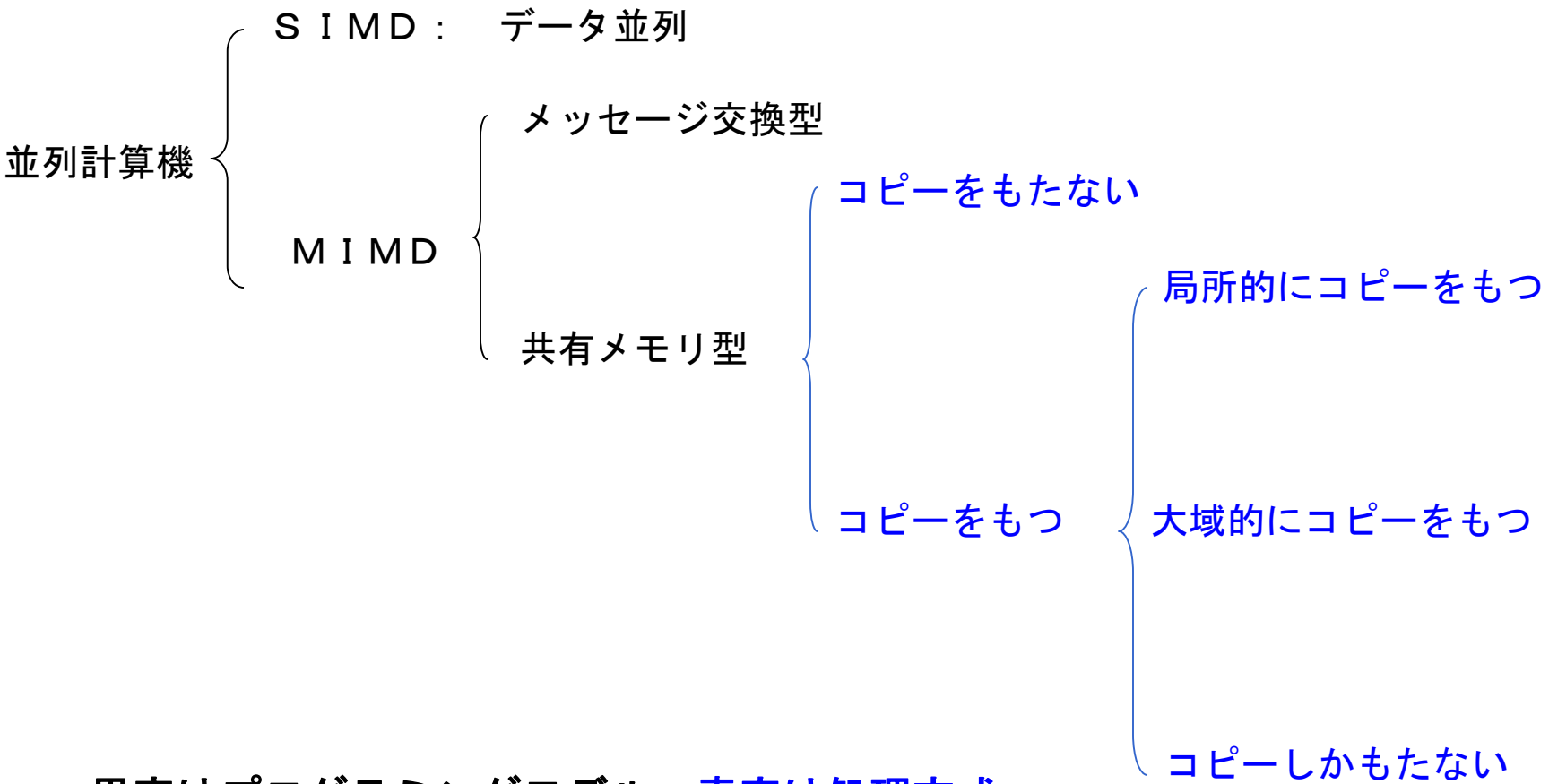
並列計算機アーキテクチャの構成上の分類

- 並列計算機
  - ＳＩＭＤ†
    - 細粒度： CM-2
    - 粗粒度： GF-11, Illiac-IV
  - ＭＩＭＤ‡
    - 大域メモリ型
      - バス結合型: SUN Fire , CMP
      - スイッチ結合型：VPP500, Cedar
    - 局所メモリ型：T3E, EM-4, RWC-1
    - 階層型（上位：局所、下位：大域）： SR8000, Origin2000, DASH/FLASH, Jump-1

† Single Instruction Stream Multiple Data Stream: 一個の命令を同時に複数のプロセッサで処理する。Data Parallelもこの範疇に入れることが多い。柔軟性はないが、問題を限れば有効。

‡ Multiple Instruction Stream Multiple Data Stream: 複数の命令を複数のプロセッサで処理する。

並列計算機のプログラミングモデル～処理方式上の分類



黒字はプログラミングモデル、青字は処理方式