同期型計算

6.1 同期(1)

複数のプロセスがお互いに先に進みすぎないようにタイミングをとる

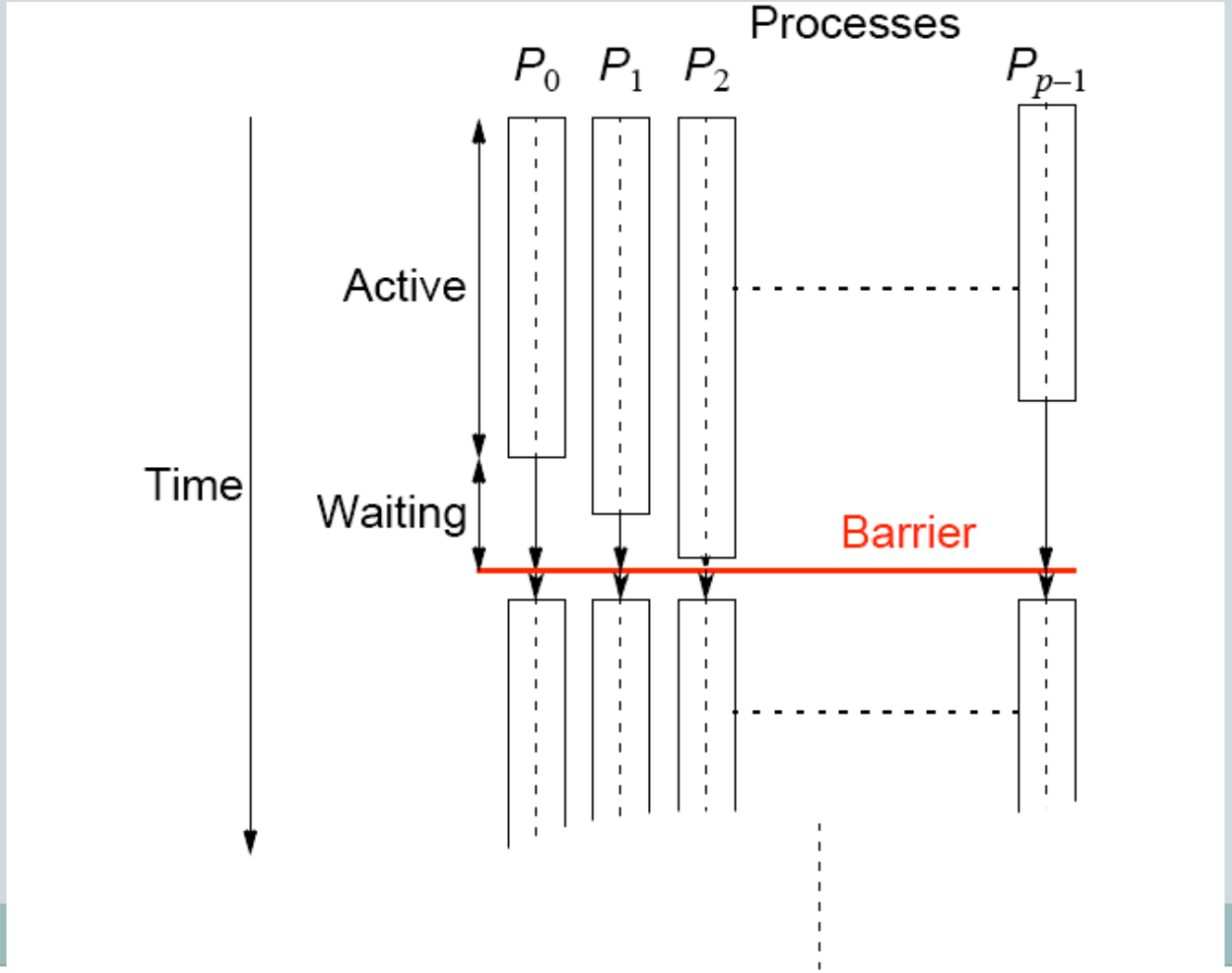
バリア(barrier)

- 各プロセスが待つべきところに挿入される.
- 各プロセスはその点に到達したら待機する.
- 全てのプロセスがその点に到達したら全てのプロセスは処理を再開する.

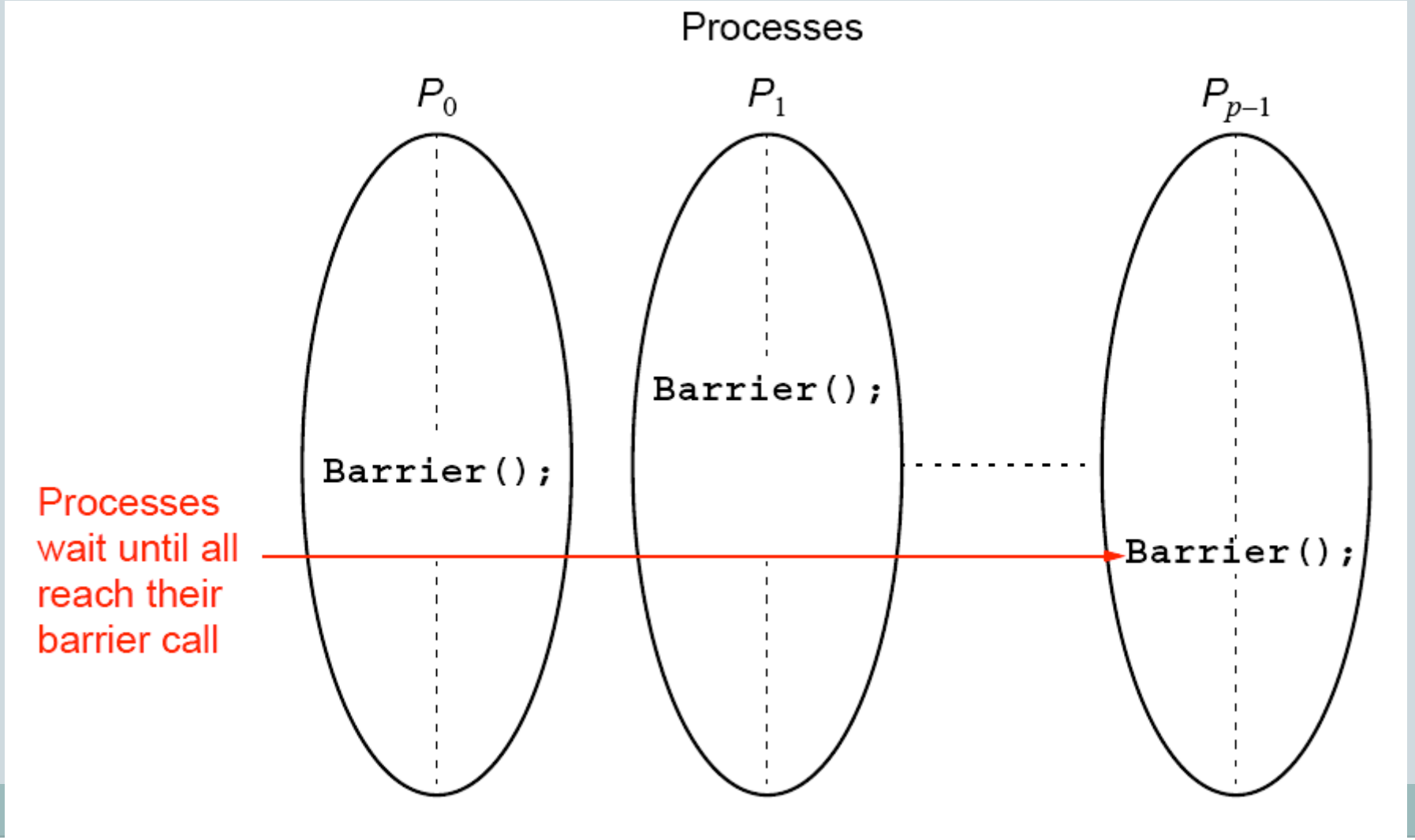
- 共有メモリ型並列システムでもメッセージ通信型システムでも実装可能
  - MPI_barrier(), pvm_barrier()

- バリアはメッセージ通信システムにおいてはコストが高くなる.
  - バリアの実装方式を理解することは, プログラミング上大事

6.1 同期(2)



6.1 同期(3)


Processes
P_0
P_1
P_{p-1}
Barrier();
Barrier();
Barrier();
Processes wait until all reach their barrier call

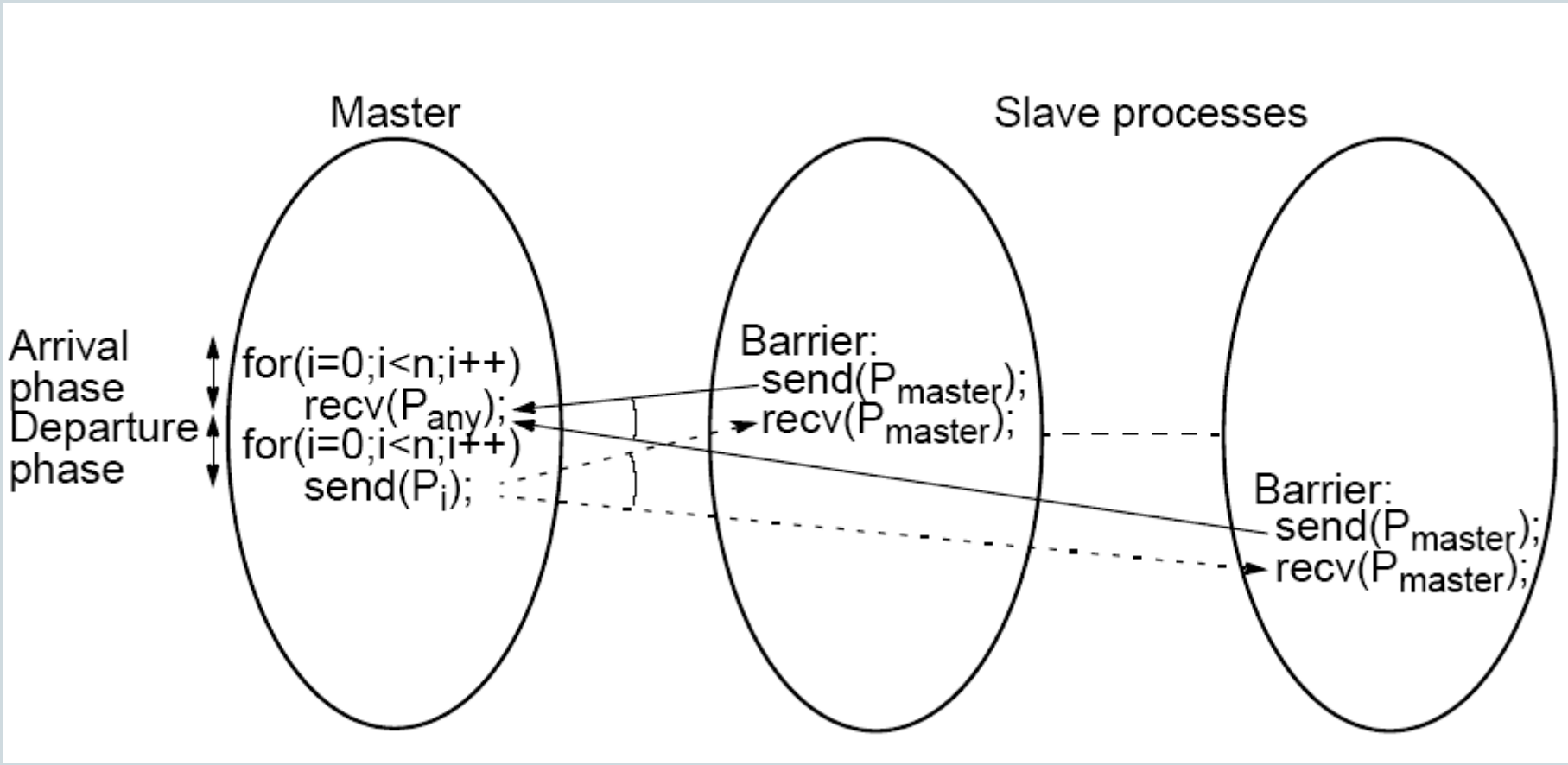
6.1.2カウンタを利用したバリアの実装(1)

カウンタを使用したバリアの実装

- カウンタでバリアに到着したプロセスの数を数える.
- 最初はゼロに初期化する.
- バリアをコールした各プロセスがカウンタをインクリメントする.
- nに到達したかをチェックし，そうであれば再開，そうでなければ待機

- 到着段階(捕獲)と出発段階(開放)
  - あるプロセスが到着段階にはいると全てのプロセスがこの段階に入るまで到着段階から抜け出せない.
- マスタプロセスがバリアカウンタを管理
  - 到着段階:スレーブプロセスがバリアに達したときに送ってくるメッセージを数える.
  - 出発段階:スレーブプロセスを開放する.

6.1.2カウンタを利用したバリアの実装(2)



6.1.2カウンタを利用したバリアの実装(3)

ブロック型send(), recv()による記述．変数iがバリアカウンタ

Master:

```
1.   for(i=0; i<n; i++)
2.   /* count slaves as they reach their barrier */
3.     recv(P_any);
4.   for(i=0; i<n; i++) /* release slaves */
5.     send(P_i);
```

Slave:

```
1.   send(P_master);
2.   resv(P_master);
```

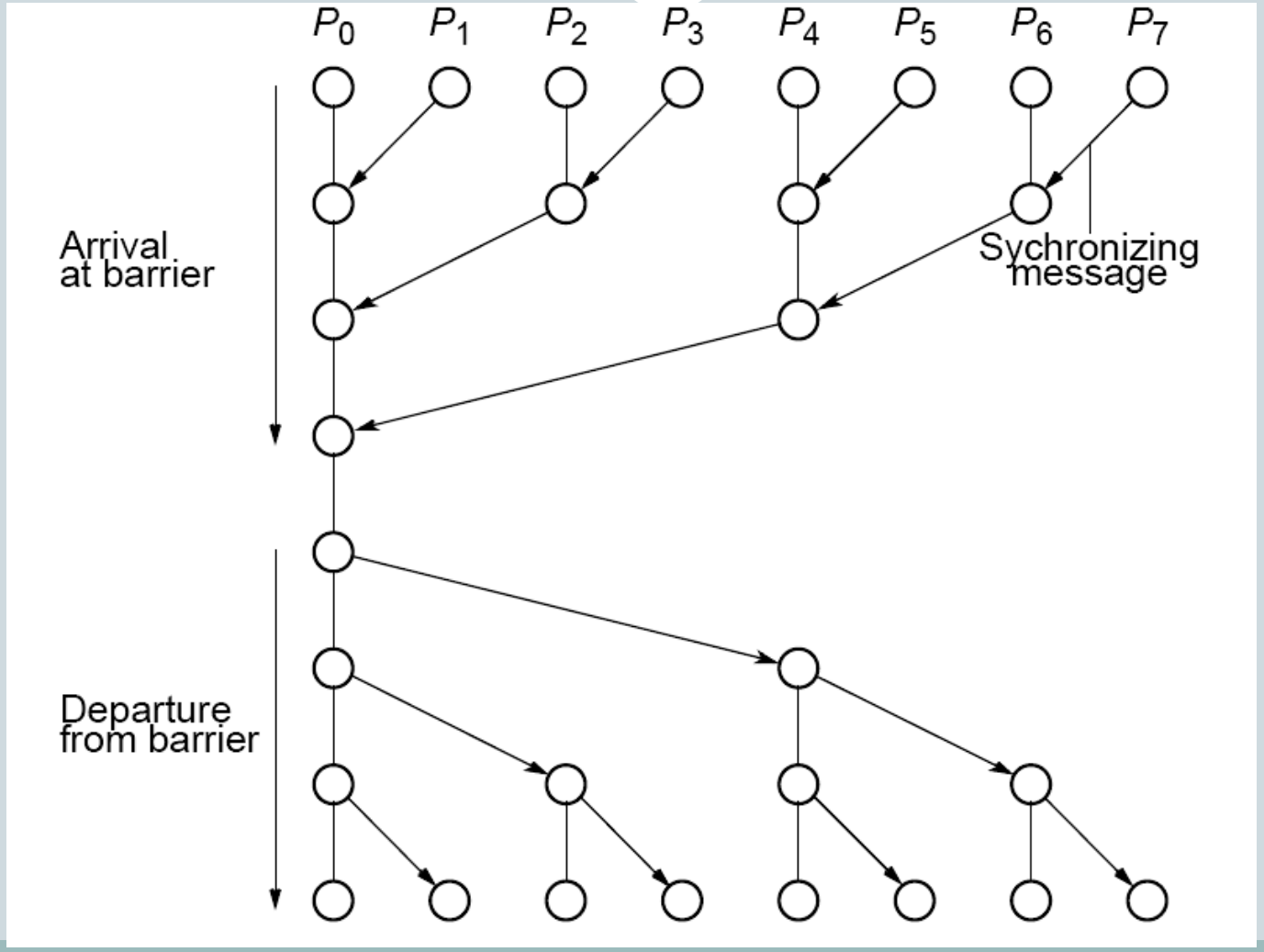
- プロセス数がn個の時，O(n)の時間複雑度

6.1.3 木構造による実装 (1)

木構造による実装

- 到着段階
  - Stage 1: P_1->P_0, P_3->P_2, P_5->P_4, P_7->P_6
  - Stage 2: P_2->P_0, P_6->P_4
  - Stage 3: P_4->P_0
  - この時点でP_0は全てのプロセスが到着したことを知る.
- 開放段階
  - Stage 4: P_0->P_4
  - Stage 5: P_0->P_2, P_4->P_6
  - Stage 6: P_0->P_1, P_2->P_3, P_4->P_5, P_6->P_7
  - この時点で全てのプロセスが開放される.
- プロセスがn個の時, O(log n)の時間複雑度

6.1.3 木構造による実装 (2)



6.1.4 バタフライを用いたバリアの実装 (1)

バタフライを用いたバリアの実装

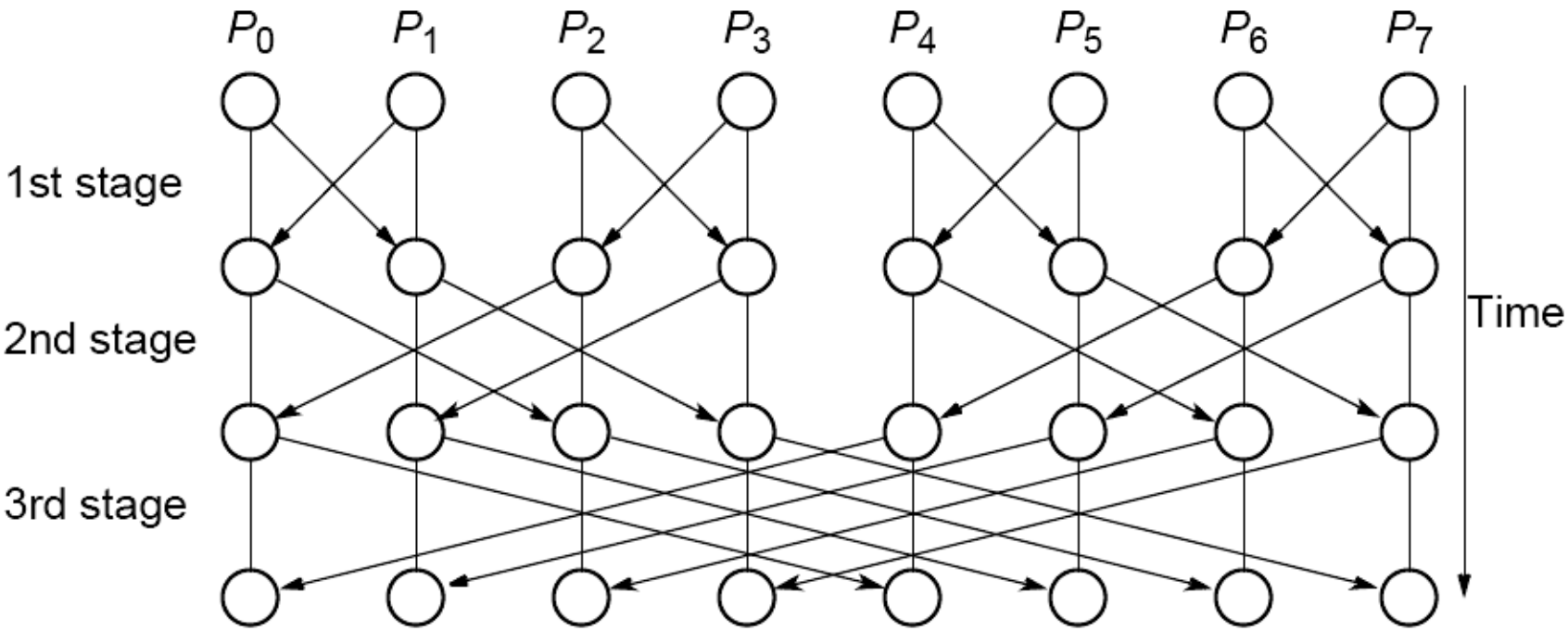
- 到着段階, 開放段階
  - Stage 1: P_0<->P_1, P_2<->P_3, P_4<->P_5, P_6<->P_7
  - Stage 2: P_0<->P_2, P_1<->P_3, P_4<->P_6, P_5<->P_7
  - Stage 3: P_0<->P_4, P_1<->P_5, P_2<->P_6, P_3<->P_7
- プロセスがn個の時, O(log n)の時間複雑度. ただし, 木構造の半分のステップ数

6.1.4 バタフライを用いたバリアの実装 (2)

1st stage $P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$

2nd stage $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$

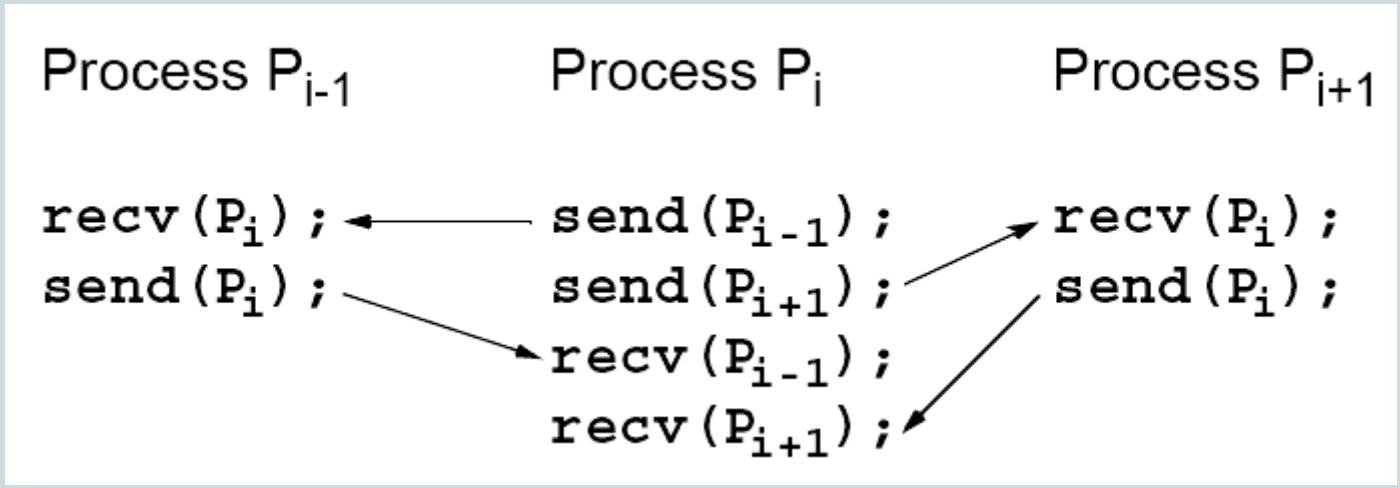
3rd stage $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$



6.1.5 局所同期

局所同期

- 全てではなく，幾つかのプロセス間で同期をとる．
  - Pi-1はプロセスPiと同期をとる
  - PiはPi+1と同期をとる



6.1.6 デッドロック

デッドロック

- 両方のプロセスが同期型ルーチンを使って，最初に送信．
  (あるいは十分なバッファを持たずにブロック型ルーチンで両方が最初に送信)
- 両方のプロセスが相手プロセスの受信を待ってしまう．
- デッドロックを回避する方法として
  - 送受信の順番を考慮したプログラミング
  - 送受信を結合したsendrecv()命令の実装

Example

Process P_{i-1}	Process P_i	Process P_{i+1}
`sendrecv(Pi);` ↔	`sendrecv(Pi-1);`	
	`sendrecv(Pi+1);` ↔	`sendrecv(Pi);`

6.2 同期型計算(1)

6.2.1 データ並列計算

- データ並列計算
  - 同期を必要とする.
  - プログラムが単純(プログラムは一種類)
  - スケーラビリティが良い.
  - SIMD(Single Instruction stream Multiple Data stream)はデータ並列コンピュータである.

6.2 同期型計算(2)

例: 配列の各要素に同じ定数を加える.

- 逐次計算

```
1.  for (i=0; i<n; i++)
2.      a[i] = a[i] + k;
```

- forall(ある並列プログラミング言語のキーワード)による記述

```
1.  forall (i=0; i<n; i++)
2.      a[i] = a[i] + k;
```

- n個の加算が同時に行われる. ループではない.

- SPMD(Single Program Multiple Data)形式のプログラム

```
1.  i = myrank;
2.  a[i] = a[i] + k;
3.  barrier(mygroup);
```

- プロセス内の計算が小さすぎるため, バリアのオーバーヘッドが大きく, 通常は, 効率的ではないことに注意.

例: プレフィックス総和計算(1)

例: プレフィックス総和計算

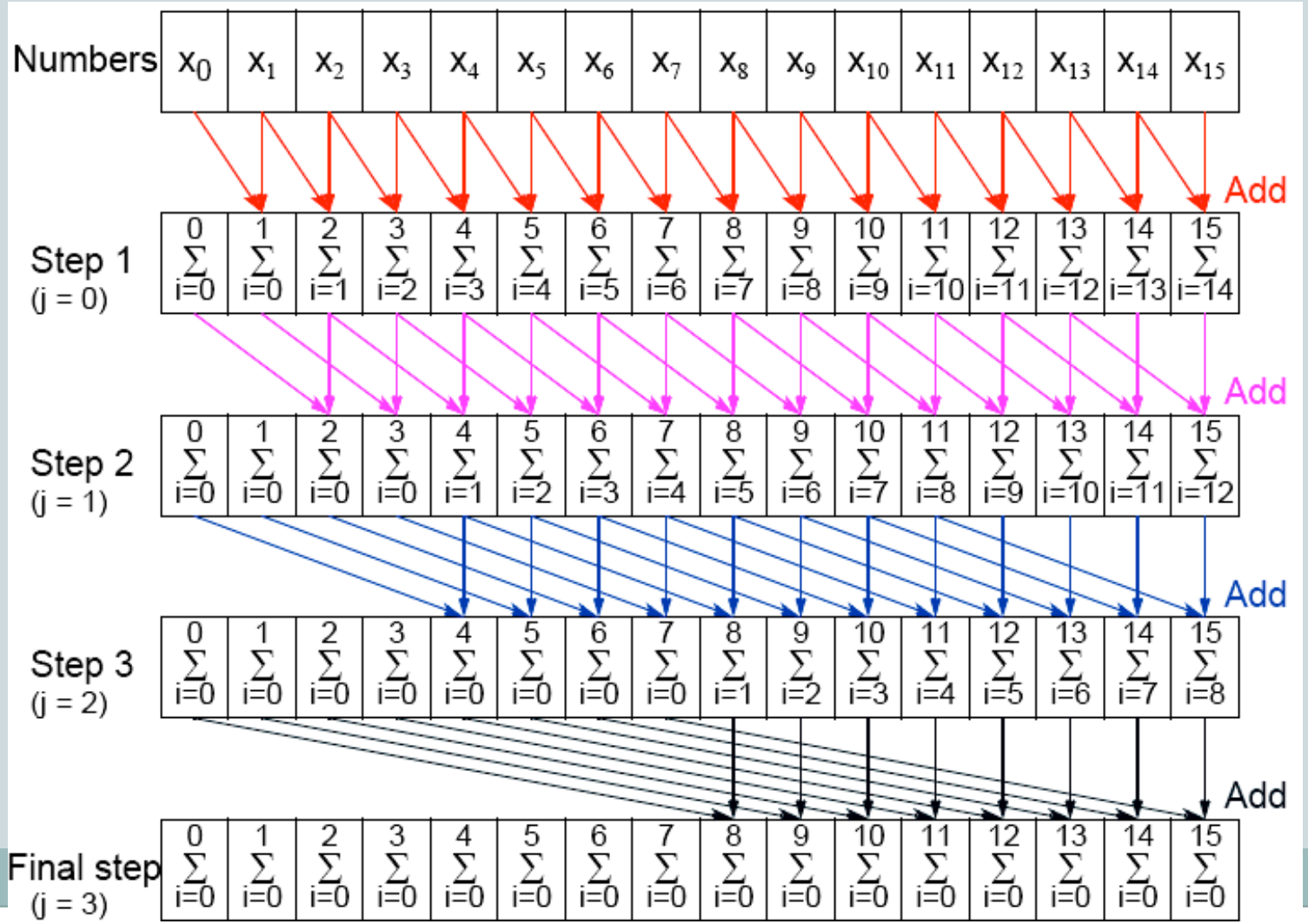
- $x_0, x_1, \ldots, x_{n-1}$が与えられたとき, 全ての部分和, $x_0 + x_1$, $x_0 + x_1 + x_2$, ... を計算する. 加算以外の演算子を使っても定義できる.

逐次計算:

```
1.  for(i=0; i<n; i++){
2.      sum[i] = 0;
3.      for(j = 0; j <= i; j++)
4.        sum[i] = sum[i] + x[j];
5.  }
```

- 計算量は$O(n^2)$.

例: プレフィックス総和計算(2)



例: プレフィックス総和計算(3)

逐次プログラム

```
1.   for(j=0; j < log(n); j++)
2.        for( i=2^j; i < n; i++)
3.          x[i] = x[i] + x[i-2^j];
```

SIMDコンピュータ用並列プログラム

```
5.   for(j=0; j < log(n); j++)
6.        forall( i=0; i < n; i++)
7.          if (i >= 2^j) x[i] = x[i] + x[i-2^j];
```

時間計算量は, O(log(n)).

6.2.2 同期式繰返し

同期式繰返し

- 同期式繰返し(同期並列)
  - 一つの繰返し中に複数の処理がある
  - それらが各繰返しの最初で同時に始められる
  - 次の繰返しは，前の繰返しの中の全ての処理が終わるまで開始されない

```
1.  for(j=0; j < n; j++){ /*for each synchronous iteration */
2.   forall(i=0; i < N; i++) /* N processes each executing */
3.        body(i);    /* body using specific value of i */
4.  }
```

- SPMDプログラムでは，

```
1.  for(j=0; j < n; j++){  /* for each synchromous iteration */
2.      i = myrank;      /* find value of i to be used */
3.      body(i);       /* body using specific value of i */
4.      barrier(mygroup);
5.  }
```

課題（並列プログラミング）

バイナリー2次計画問題(BQP)の近傍探索を並列化せよ.

BQPの定義

- 入力:
  - n x nの行列Qが入力として与えられる.
- 問題:
  - 次の目的関数を最大化するサイズnのベクトルxを求めよ.

    $f(x) = x^t Q x,\ \ Q(i, j) \text{ in 整数}, x(i) \text{ in } \{0, 1\}$
  - xは二値ベクトル，入力Qはn×nの整数行列

近傍探索の設計

- xの近傍領域
  - xはサイズnの2値ベクトルである.
  - xの近傍集合N(x)を次のように定義する.

    $N(x) = \{x' \mid h(x, x')==1\}$
  - ただし，h(x, x')はxとx'のハミング距離を表す.

課題(並列プログラミング)(2)

近傍探索アルゴリズム

```
1.    Local-Search(x){
2.        do{
3.          F[0] = f(x);
4.          max = 0;
5.          for (i=1; i<=n; i++){
6.            x_i = x reversed i-th bit
7.            F[i] = f(x_i);
8.            if (F[i] > F[max]) max = i;
9.          }
10.         x = x reversed (max)-th bit;
11.       }while(F[max] > F[0]);
12.       x = x reversed (max)-th bit;
13.   /* F[0] = f(x) is the local optima */
14.     }
```

並列化のヒント:

- データ分割
- 同期計算