コンピュータシステムII（6）

情報基盤センター
天野　浩文

1

前回のおさらい（1）

- 並列処理のやり方…何と何を並列に行うのか
  - **コントロール並列プログラミング**
    - 同時に実行できる多数の処理を，多数のノードに分配して同時に処理させる
    - しかし，「同時に実行できる多数の処理」を見つけるのは難しい
  - **データ並列プログラミング**
    - 大量のデータを多数の演算ノードに分配して，それらに同じ演算を同時に適用する
    - コントロール並列よりも，多数の演算ノードを利用しやすい

2

前回のおさらい（2）

- 並列に実行できる機械語コードを得る方法
  - **逐次型プログラムの自動並列化**
    - 逐次型プログラミング言語で記述されたプログラムを，コンパイラの力で（自動的に）並列化する．
  - **並列処理記述言語によるプログラム作成**
    - 逐次型プログラミング言語を並列処理記述用に機能拡張した言語で書かれたプログラムを，コンパイラが並列化する．
  - **逐次型プログラムの書き換え**
    - 逐次型プログラミング言語で記述されたプログラムを，（人間が）並列処理用に書き換え，それに従ってコンパイラがプログラムを並列化する．

3

前回のおさらい（3）

- 逐次型プログラムの自動並列化の利点
  - プログラマがコントロール並列やデータ並列を意識する必要すらない．
  - 既存のプログラムがそのまま並列化できる
    - **このような技術が本当に完成すれば理想的**
  - 並列計算機のアーキテクチャやOSに全く依存しないプログラム開発が可能
- 欠点
  - 現在のコンパイラ技術では，自動的に並列化できる部分は限られている．
    - 「この部分は並列化しても正しい結果となる」ということが，機械的に判断できないことも多い．
  - 自動並列化コンパイラは，現在も重要な研究開発課題となっている．

4

前回のおさらい（4）

- 並列処理記述言語
  - 既存の逐次型言語を拡張して，並列処理の記述に適した言語を作る．
    例：HPF（High-Performance FORTRAN）
  - 全く新たな言語を作ることもあるが，あまり普及した例はない．
- 並列処理記述言語によるプログラム作成の長所
  - ターゲットとなる並列計算機のアーキテクチャやOSの機能を活用した並列処理が行いやすい．
- 短所
  - 並列処理することを念頭においてプログラムを書かなければならない（逐次型プログラムよりは難しい）
  - 同じ言語を多種多様な並列計算機で利用できるようにするのは大変．

5

前回のおさらい（5）

- **逐次処理用に書かれたプログラムを人間が書き換える**方法
  - プロセス間通信や同期のための特殊な手続き呼び出しを書き入れる．
    - 例：MPI
  - 並列化可能な（＝コンパイラに並列化させたい）部分を示す特殊な構文で，コンパイラに対する指示を書き加える．
    - 例：OpenMP
- 逐次型プログラムの書き換えの長所
  - 自動並列化よりは実行性能が期待できる．
  - 新たな言語よりは，プログラマ・計算機メーカに普及させるのが容易．
- 短所
  - 並列処理することを念頭においてプログラムを書かなければならない
    - 逐次型プログラムよりも，書くのがかなり難しい．
    - 並列処理記述のための言語を用いる場合と比べても，書く上でやや注意（工夫）を要する部分がある．

6

前回のおさらい（6）

- MPI（Message Passing Interface）
  - 並列に動作するプロセス間の通信や同期に必要な関数（手続き）をまとめたライブラリの仕様（規格）
    - 新しい言語ではない．
    - プログラマは，逐次型プログラムの中に，これらの関数の呼び出しを適切に埋め込む．
      - しかし，もはや逐次型プログラムではなくなる．
    - 分散メモリ型・共有メモリ型いずれの並列計算機でも利用できる．
  - 現在のところ，C/C++とFortranで使うことができる．
    - それぞれの**言語の文法はまったく変更されていない**ので，既存のコンパイラでコンパイルできる．
    - ただし，コンパイルして生成される機械語プログラムは並列実行用のものになる．

7

MPIプログラムの概観（Cの場合）

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char* argv[]) {
  int my_rank;
  int p;
  ...

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  ...

  MPI_Finalize();
}
```

並列に実行される部分

8

前回のおさらい(7)

- OpenMP
  - 一種の「プログラミングスタイル」とでも呼ぶべきもの
    - コンパイラに並列化の箇所および方法を指示するための**特殊なコメント文・指示文の文法**
    - コンパイルされたプログラムが実行されるときの環境を決定したりその情報を取得したりするしくみ
    - 並列実行を補助するための特殊なサブルーチンや関数
  - OpenMPに対応した並列化機能を有するコンパイラが，並列実行可能な機械語プログラムを生成する．
    - 現在のところ，共有メモリ型並列計算機でのみ利用可能．
    - 現在のところ，C/C++とFortranのみ．
    - 並列実行補助用の特殊な関数やサブルーチンを使っていなければ，並列化の機能を持たないコンパイラでも，逐次型プログラムとしてコンパイルできる．

9

OpenMPプログラムの一例(Cの場合)

```
#include <stdio.h>
#include <omp.h>
main(){
  int i;
  double z[100], x[100], a, b;

  /* x, a, b の初期化など */

  #pragma omp parallel for

  for (i = 0; i < 100; i++) {
    z[i] = a*x[i] + b;
  }

  /* z の出力など */
}
```

「この直後の **for** 文を複数のスレッドで並列に実行せよ」という指示

100回の繰り返しを，環境変数 **OMP_NUM_THREADS** で指定された数のスレッドで分担して並列実行する．

- Cでは，**pragma**文(コンパイラ指示文)で直後の文の並列化指示を書く．
- これらの指示を「解釈」できないコンパイラは，それを無視する．

10

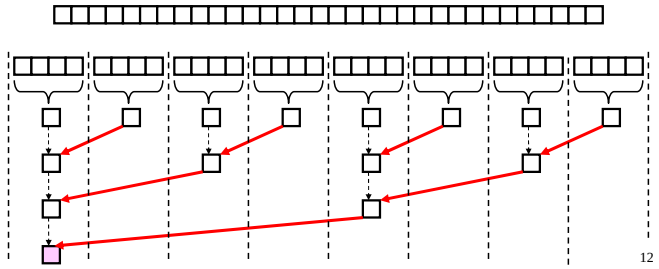
プロセス間通信

現在，教科書で取り扱っている順序を大きく離れて迂回している．
前々回に3章の「3.1.1 プロセスとスレッド」のところを扱った後，前回は「7.1 並列化コンパイラと並列処理記述言語」に迂回した．
今回は「5. プロセス間通信機構」と「3.2 メモリ」を扱う．

11

具体的な並列処理の例(1)

- 長さ N の一次元配列の総和や最大値を P 個のプロセス(またはスレッド)で計算する．
  - 簡単のため，N は P で割り切れるものとする．
  - 配列を P 個に分割し各プロセスが部分的な結果を計算する．
  - その結果を代表のプロセスに集めてくる．



12

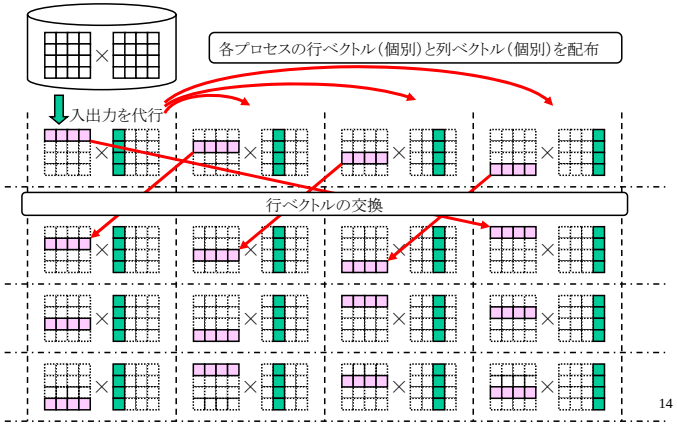
具体的な並列処理の例（2）

- P 個のプロセス（またはスレッド）が持つデータを1カ所に集めてくる操作は，さまざまな並列処理プログラムでしばしば現れる．
  - いちいち，1対1のやりとりを P 回繰り返すようにプログラムを記述するのは面倒．
- P 個のプロセス（またはスレッド）からデータを集めてくるが，その計算結果は，代表だけではなく全体で共有したい，ということもある．
  - 全体の総和・総積・最大値・最小値・ビット和・ビット積などの結果を得る操作も，さまざまな並列処理プログラムでしばしば現れる．

13

具体的な並列処理の例（3）

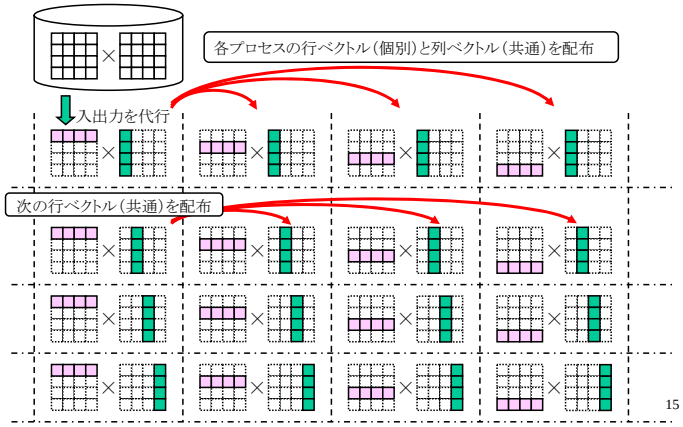
- $N \times N$ の行列の積を，P 個のプロセスで計算する．



14

具体的な並列処理の例（4）

- $N \times N$ の行列の積を，P 個のプロセスで計算する．



15

並列処理におけるデータの受け渡しの必要性

- 並列処理においては，どうしても他のプロセス（スレッド）とデータを受け渡さなければならないことがある．
  - 例（1）～（4）の実線矢印で記述したデータの受け渡しは，それぞれの問題を解くのに必須である．

16

並列処理に頻出するデータ受け渡しのパターン

- 1対1
  - ある単一のプロセス(スレッド)が持っているデータを，別の単一のプロセス(スレッド)に渡す．
  - この操作を必要な回数だけ繰り返せば以下のデータ受け渡しもすべて実現できる．
- 1対 *N*
  - ある単一のプロセス(スレッド)が持っている同じデータを，他のすべてのプロセス(スレッド)に渡す．
- *N* 対1
  - 他のすべてのプロセス(スレッド)が持っているばらばらのデータを，ある単一のプロセス(スレッド)に集めてくる．
  - 集めてくるだけでなく，最大値等の計算も合わせて行うことが多い．
- *N* 対 *N*
  - すべてのプロセス(スレッド)が持っているデータを集めて計算した結果を，全プロセス(スレッド)が共有する．

前回のMPIの通信関数の説明を読み返してみると，具体的なイメージをつかむのに役立つ．

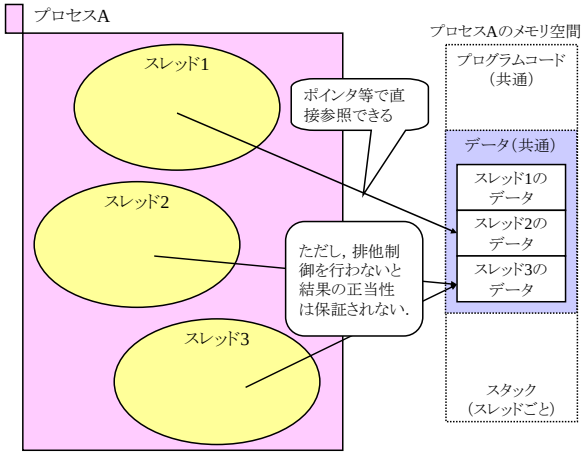
17

並列処理におけるデータ受け渡しの実現法

- スレッドどうしでデータを受け渡すには…
  - メモリ空間を共有しているので，他のスレッドの変数に直接アクセスできる．
  - 他のスレッドの変数をメモリ上の番地により直接参照できる．
    - ただし，書き込み時の排他制御は必要
- プロセスどうしでデータを受け渡すには…
  - メモリ空間が共有されないので，他のプロセスのデータに直接アクセスすることはできない．

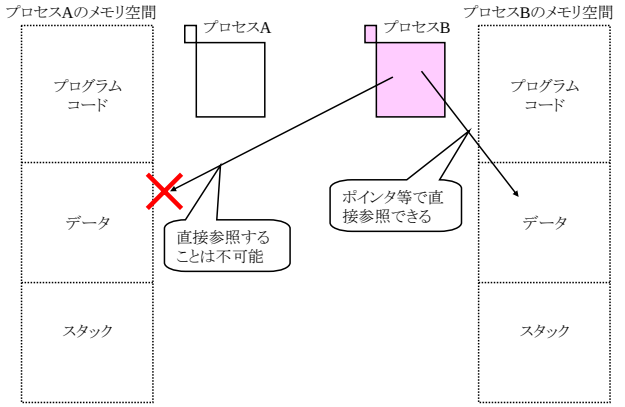
18

スレッド間のデータ受け渡し



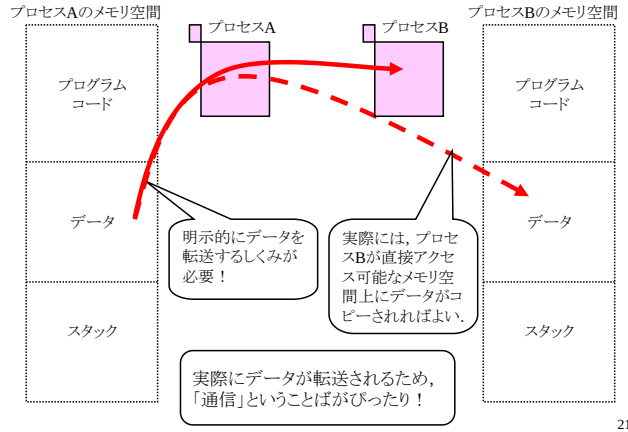
19

プロセス間のデータの受け渡し（1）



20

プロセス間のデータの受け渡し（2）



21

プロセス間通信の実現法

- 「プロセスどうしはメモリ空間を共有しない」という原則を守る場合
  - プロセス間通信は，異なるメモリ空間の間のデータのコピーとして実現しなければならない．
    - ただし，データのコピーにはそれだけの処理時間がかかるので，非常に高速な通信を行うのには不適．
- プロセス間通信のためのデータのコピーを避けるためには
  - **プロセスのメモリ空間の一部または全部が重なっていてもよいことにする．**
    - コピーは不要になる．
    - しかし，プロセスのメモリ空間の「保護」は難しくなったり，不可能になったりする．

このあたりから，教科書の「プロセス間通信機構」の解説につながっていく．

22

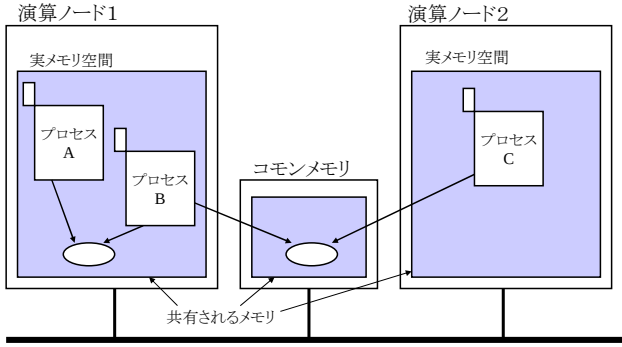
コピーを行わないプロセス間通信（1）

- 共有メモリを経由したデータ受け渡し
  - 物理的な共有メモリがある場合には特に有効
    - 実記憶
      - 仮想記憶を用いない
    - 単一仮想記憶
      - 仮想記憶空間の全体が共有される
    - 多重仮想記憶
      - 仮想記憶空間の一部が共有される

23

実記憶によるプロセス間通信

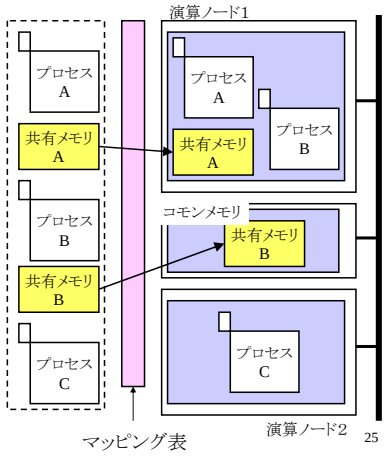
- 仮想記憶機構がなく，実メモリのすべてがどのプロセスにも見えている．
  - ただし，受け渡すべきデータはコピーしなくてすむ場所に配置すること．
  - 全プロセスで共有するデータはコモンメモリに



24

単一仮想記憶によるプロセス間通信

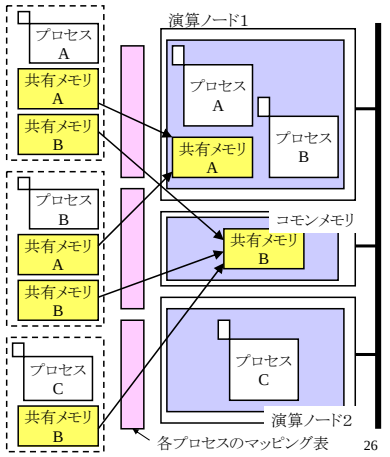
- 仮想記憶はあるが，それを全プロセスで共有
  - 仮想記憶空間の番地と実メモリ上の番地を対応付けるマッピング表を活用する．
  - ただし，受け渡すべきデータはコピーしなくてすむ場所に配置すること．
  - 全プロセスで共有しなければならないデータはコモンメモリに



25

多重仮想記憶によるプロセス間通信

- 各プロセスの仮想記憶空間の一部が重なっている
  - 単一仮想記憶の場合と同様に，各プロセスのマッピング表を活用する．
  - ただし，受け渡すべきデータはコピーしなくてすむ場所に配置すること．
  - 全プロセスで共有しなければならないデータはコモンメモリに



26

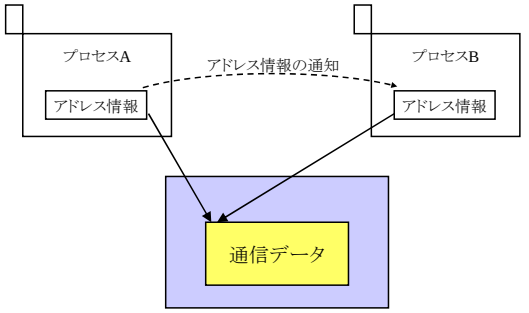
メッセージ

- プロセス間でデータを受け渡すもう一つの方法
  - 送り手と受け手がそれぞれ固有の操作を行う
    - 受け手（のプロセスID）を指定してメッセージを送る（SEND）
    - 送り手（のプロセスID）を指定してメッセージを受け取る（RECEIVE）
  - 「通信」のイメージに近いプログラミング
    - ただし，本当にデータのコピーが必要かというと，必ずしもそうではない．

27

コピーを行わないプロセス間通信（2）

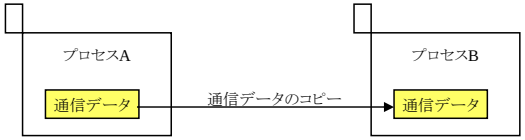
- アドレス渡し法
  - プロセス間で共有されるメモリ空間がある場合，送るべきメッセージをそこに置き，そのアドレス情報だけを受け手に通知すれば，受け手は実際の通信データに直接アクセスできる．



28

コピーを行うプロセス間通信

- メッセージ複写法
  - プロセス間でメモリ空間が共有できない場合には，通信データを実際に転送(コピー)するしか方法がない．
    - その分だけ，余計に時間がかかる．



  - 実装には，OSに用意されている**ソケット**などを用いる．

ソケットの詳細については，「クライアント・サーバシステム」のところで扱う予定．

29

コピーの有無によるプロセス間通信の比較

データ受け渡しの方法	長所	短所
共有メモリ および アドレス渡し	送受信が非常に速い(データのコピーが不要なので)	① プロセス間で共有できるメモリ空間が必要 ② 送受信するデータのアクセス保護が弱い(送信後も送り手が内容を変更できたりする)
メッセージ複写	送受信するデータのアクセス保護が強い(送信後，受け手しか内容を変更することができない)	送受信に時間がかかる(データのコピーが必要なので)

30

プロセス間通信のまとめ

- プロセス間通信が必要となる場合
  - 並列処理(分散処理)においては，処理を進める上で他のプロセス(スレッド)が保有するデータを必要とすることが多々ある．
- プロセス間通信のパターン
  - 1対1，1対 N，N 対1，N 対 N など．
- (標準的な)プロセスとスレッドの通信の違い
  - メモリ空間の共有の有無
- プロセス間通信を効率化するための工夫
  - プロセスのメモリ空間の一部または全部を共有させる
- プロセスのメモリ空間を共有させられない場合は
  - メッセージ複写法

31

分散共有メモリ

32

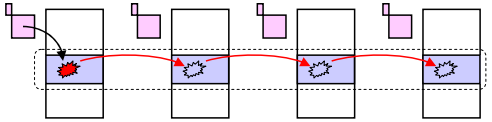
分散共有メモリ（1）

- 分散メモリ型並列計算機，クラスタシステム，広域分散システムでは，全演算ノードが共有するメモリは存在しない．
  - 各演算ノードで動作するプロセスがデータを共有したい場合には…
    - 通信機能によってデータをやりとりする必要がある．
    - プログラミングが面倒
      - ローカルメモリにあるデータにアクセスする場合と，リモートメモリにあるデータにアクセスする場合とで，プログラムの書き方が大きく異なる．
  - ハードウェア的な共有メモリが存在しなくても，ソフトウェアでそれを実現できないか？
- **分散共有メモリ**（distributed shared memory）
  - 仮想共有メモリ（virtual shared memory）と呼ぶこともある．

33

分散共有メモリ（2）

- プログラムから見た分散共有メモリの「見え方」
  - プログラミング言語の文法に，他のプロセッサ上の変数の値を直接読んだり，それに直接代入したりできるような拡張機能を追加するもの
    - 例：「5番のプロセッサの変数 x に代入する」などという構文を新たに追加する．
  - メモリ空間の一部分を「共有部分」として扱うもの
    - 例：「1001番地から2000番地は共有」，あるいは，「この変数は共有」と決めておき，そこにデータを書くと，他の全ノードに変更を反映させるようにする．



34

分散共有メモリ実現方式の分類

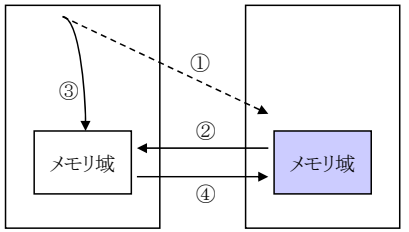
- **コンパイラ方式**
  - コンパイル時に，リモートメモリへのアクセスを通信処理に書き換えて，実行可能プログラムを生成する．
  - プログラミング言語の文法を拡張するため：
    - リモートアクセスをある程度意識することが必要．
    - 新しいコンパイラが必要．
    - 既存のアプリケーションプログラム（AP）は再コンパイルが必要．
- **実行時方式**
  - プログラムの実行時に分散共有メモリへのアクセスがおきると，例外（exception）を発生させて制御をOSに移す．
    - OSが，リモートメモリへのアクセスを代行する．
  - 言語の拡張は不要：
    - 既存のAPはそのまま使える．
    - 実現する機構はやや複雑．OSの改造が必要．

35

リモートメモリの操作方式（1）

- **複写方式**

① 遠隔の演算ノードのメモリへのアクセスが発生する．
② 操作対象のメモリ域が複写されてくる．
③ 複写されたメモリ域を操作する．
④ 更新があれば，更新内容を他の演算ノードに反映させる．

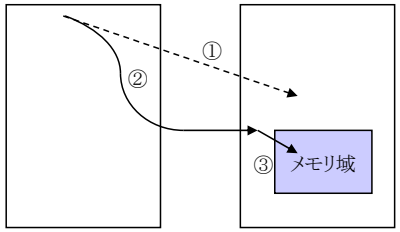


一時的にせよ，同じデータが複数の場所にあることになるため，一貫性を保つしかけが必要になる．

36

リモートメモリの操作方式（2）

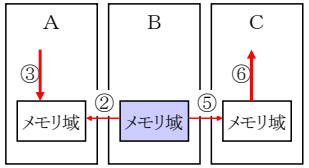
- **非複写方式**
  - ① 遠隔の演算ノードのメモリへのアクセスが発生する.
  - ② メモリ操作の内容を，そのメモリのある演算ノードに転送する.
  - ③ 転送された操作内容に基づいてメモリ操作を行う.



37

複写方式においてアクセスが競合する場合

- 以下のようなタイミングで演算ノードBのメモリ域へのアクセスが発生すると…
  - ① 演算ノードAで書き込み要求
  - ② 演算ノードBからAへ複写
  - ③ 演算ノードAで書き込み
  - ④ 演算ノードCで読み取り要求
  - ⑤ 演算ノードBからCへ複写
  - ⑥ 演算ノードCで読み取り



- Aでの書き込みがBに反映されるタイミングによって…
  - ③以降⑤以前：Cは時系列通りの「正しい」内容を読める.
  - ⑤以降：Cの読む内容は「正しくない」.
- 「正しさ」，言い換えると，一貫性（consistency）をどう定義するか？
  - 並列計算機や広域分散システムでは，①～⑥がどのような順序で起きるか，わからない（制御できない）こともある.
  - ただし，個々の演算ノード内部で起こる操作の順序関係はわかる.

38

メモリの一貫性制御方式（1）

- **逐次（sequential）コンシステンシ（SC）**
  - 一貫性の定義
    - 並列プログラムの実行結果が，単一ノード上で時分割実行されたときと同じであること.
      - 各演算ノード内で起こる操作の順序を保存し，同じデータに対する複数の操作が同時に起こることを許さない.
  - 実現の方法
    - 書き込みが起きたら，それが反映されるまで，次の読み込みを許可しない.
      - 書き込みが発生するたびに共有メモリ域の書き戻し処理を行い，それが終了するまでは，同じメモリ域からの複写を禁止あるいは保留する.
  - 問題点
    - 大量のデータ転送が発生するので効率が悪い.

39

メモリの一貫性制御方式（2）

- 共有データに対するアクセスを排他的に行う場合は，書き込みのたびに書き戻す必要はない.
  - 共有データに対するアクセスの前には，必ず，排他的なアクセス権を獲得する（Acquire）.
  - 操作（何回連続してもよい）が終了すると，アクセス権を解放する（Release）.
  - AcquireとReleaseの間の操作は，他の演算ノードと競合しない.
- **リリースコンシステンシ（release consistency, RC）**
  - Release時に1回だけ書き戻し処理を行う.
    - SCよりもデータ転送回数が削減される.

40

メモリの一貫性制御方式（3）

- レイジーリリースコンシステンシ（lazy release consistency, LRC）
  - Release 時ではなく，次のAcquire 時に書き戻しを行う．
  - 共有メモリ域全部を書き戻すのでなく，更新の前後の差分情報のみを記録しておき，書き戻しの際には，更新のあった箇所の差分情報のみ転送する．
    - データ転送は削減できる．
    - 差分情報を生成したり，それを適用する処理には大きな負荷がかかる．
    - 差分情報を保持する分，よけいにメモリが必要になる．

41

メモリの一貫性制御方式（4）

- HLRC（home-based lazy release consistency）
  - AcquireとReleaseを用い，差分情報を用いる点はLRCと同じ．
  - 共有メモリ上の各ページに，それを管理する特定の演算ノード（home）を割り当てる．
    - 差分情報はRelease時にhomeに転送し，homeでは即時に反映させ，最新の版を保持する．
    - 他のノードは，Acquire時に，homeから最新の内容を取得する．
  - 差分情報を保持する期間は短くなるので，LRCよりも使用するメモリは少なくてすむ．
    - 差分情報の生成と適用にかかる負荷は同じ．

42

分散共有メモリのまとめ

- ハードウェア的には存在しない共有メモリを，ソフトウェア的に実現する方式
  - 実現方式
    - コンパイラ方式と実行時方式
  - リモートメモリの操作方式
    - 複写方式と非複写方式
  - 複写方式においては，一貫性制御が必要
    - SC, RC, LRC, HLRC

43