コンピュータシステムII（13）

情報基盤センター
天野　浩文

1

演習問題の回答例　1

1. 以下の2つのプログラム片はセマフォに対するPV操作の概要を示している．しかし，危険区間へのアクセスの前後にすべてのプロセス（スレッド）がこのPV操作を実行したとしても，排他制御が失敗する可能性が残っている．それはどのような場合かを説明せよ．

```
P(S): while S<=0 do {};
      Sを1減らす
```

```
V(S): Sを1増やす
```

P操作の中のセマフォSの値を検査してからSの値を減じるまでの間に，他のプロセス（スレッド）のP操作のSの値の検査が行われてしまうと，どちらのプロセス（スレッド）も，危険区間に同時に入れてしまうことがある．

2

演習問題の解答例　2

2. デッドロックが起こるための4つの必要条件を説明せよ．

- **相互排除**
  少なくとも1つの資源は，同時にひとつのプロセス（スレッド）にしか利用できない．
- **確保と待機**
  少なくとも1つの資源を確保しているが，他のプロセス（スレッド）が先に確保してしまった資源が利用可能になるのを待っているプロセス（スレッド）が少なくとも1つある．
- **非プリエンプション**
  いったん確保された資源は，そのプロセス（スレッド）の処理が終了して解放されるまでは利用可能にならない．
- **巡回待機**
  「プロセス（スレッド）i はプロセス（スレッド）j の確保している資源が解放されるのを待っている」という関係が，$p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n \rightarrow p_1$という形式で閉路を形成している．

3

演習問題の解答例　3

3. デッドロックを検出する方法を2つあげ，それぞれ説明せよ．

- **待ちグラフを用いる方法**
  「プロセス（スレッド）p_i はプロセス（スレッド）p_j の確保している資源が解放されるのを待っている」という関係を，$p_i \rightarrow p_j$という枝（edge）で表したグラフを用いて，これに閉路があるときには「デッドロック状態にある」，ないときには「デッドロック状態にない」と判定する方法．
- **時間切れ方式**
  ある程度の時間が経過しても終了しない処理は，デッドロックに陥っていると推定する方法．ただし，単に時間がかかっているだけのものをデッドロックと誤判定する危険もある．

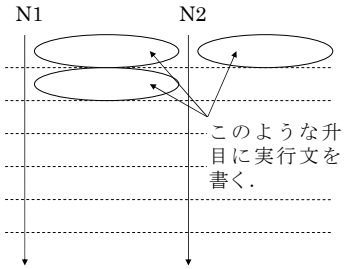
4

演習問題 4

4. 左下の2つのトランザクションT1, T2がそれぞれ演算ノードN1, N2で実行されるものとする. これらが隔離性(直列可能性)を満たす実行スケジュールの例と満たさない実行スケジュールの例を, それぞれ右下のようなタイムチャート上に表せ. ただし, いずれの実行文も1クロックサイクル(タイムチャート上の升目1つで実行できるものとする.

```
T1:
READ(A);
READ(B);
B=A+B;
WRITE(B)
```

```
T2:
READ(B);
B=B+100;
WRITE(B);
```

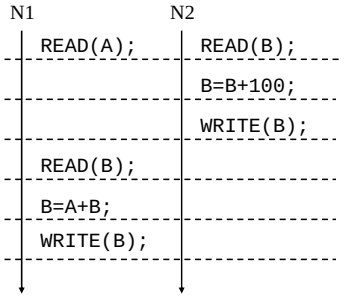


5

回答例 4

- 考え方
  - T1→T2またはT2→T1の順で直列に実行した場合の結果と同じになるスケジュールと, いずれとも異なる結果になるスケジュールを作ればよい.

満たす例



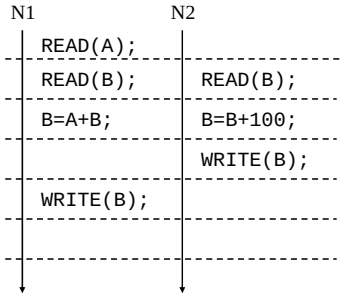
同じデータを読んでから書き込むまでの区間が重ならないようにすると, うまくいくことが多い.

6

回答例 4(つづき)

- 考え方
  - T1→T2またはT2→T1の順で直列に実行した場合の結果と同じになるスケジュールと, いずれとも異なる結果になるスケジュールを作ればよい.

満たさない例



同じデータを読んでから書き込むまでの区間をオーバーラップさせると, 直列可能でないスケジュールになりやすい.

7

演習問題の解答例 5

5. 二相コミットについて説明せよ.

分散トランザクション処理において, トランザクションの原子性を保証するために, 各演算ノードでの計算が終了した後のコミット操作を, 以下の2つの段階に分割して実行すること.

- –第一段階
  - •各演算ノードでコミットの準備ができたかどうかを確認する.
- –第二段階
  - •各演算ノードのリソースマネージャからの「投票」を受けて, コミット／アボートの判断を行い, 各ノードでコミットを行う.

8

演習問題の解答例 6

6. クライアントサーバシステムとピアツーピアシステムについて，両者の違いがよくわかるように説明せよ．

- クライアントサーバシステム
  サーバ（あるサービスを提供するプログラム）とクライアント（そのサービスを受けるプログラム）によって構成されたシステム
- ピアツーピアシステム
  サービスを提供するものとサービスを受けるものの関係を固定せずに，状況に応じてサービスを提供したりサービスを受けたりするプログラム群によって構成されたシステム

9

並列アルゴリズム設計の指針

ここで紹介する方法は，決して唯一絶対のものではないが，非常にわかりやすくまとめられており，将来実際に並列アルゴリズムや並列プログラムを設計する際にきっと役立つであろうと考える．
出展：Ian Foster著: “Designing and Building Parallel Programs”, Addison-Wesley, 1995.

期末試験の出題範囲には含めない．

10

大前提

- 解くべき問題が与えられたとき
  - **逐次型処理でその問題を解く方法はわかっている（既知である）ものとする**．
    - なぜなら，逐次型（非並列）処理でも解く方法がわからない問題を，並列処理によって解くことはできないから．
  - **その問題を並列に解くことのできる方法は，必ずしも一通りではない**．
    - 元になる逐次処理アルゴリズム自体も，必ずしも一通りではない．
    - ただし，処理に用いる並列計算機の構成・機能・規模あるいはプログラム作成の手間などの要因を検討することにより，どれが最も望ましいかを選ぶことはできるだろう．

11

Fosterの言う『並列アルゴリズム設計の秘訣』

- ある問題が与えられたときに，それを並列に解く方法を，なるべく後で「軌道修正」のしやすい形で，もし複数あるならそれらをすべて，列挙するところから始める．
  - 実行する**並列計算機の構成や機能に依存するような事項の検討は，なるべく後回し**にする．
  - 運悪く途中でまずい選択をしてしまったとしても，後戻りのコストがなるべく小さくなるようにする．
- あまり早い段階で，可能な選択肢を捨てないようにする．
  - 最終的に一番よいものが選べるようにする．

12

設計手順の概要

- **分割**(partitioning)
  - 実行すべき計算と処理すべきデータを，小さな「仕事」(タスク，task)に分割する．
    - この「タスク」は，プロセスの同義語ではない．
- **通信**(communication)
  - 各タスクを実行する際に，どのような通信(情報の交換)が必要になるかを考える．
- **凝集化**(agglomeration)
  - 得られたタスク集合とそれらの間の通信構造について，実行計算機やプログラム開発コストを考え，より大きなタスクにまとめていく．
- **割り付け**(mapping)
  - 各タスクをそれぞれ1つのプロセッサに割り当てる．

13

1．分割設計

- なるべく小さなタスクを，できるだけたくさん考える．基本的な方法は2つ．
  - データを分割する方法(**領域分割**，domain decomposition)
  - 計算を分割する方法(**機能分割**，functional decomposition)
- 普通の人は，領域分割のほうが考えやすいかもしれない．
- どちらの方法を採るにせよ，**この段階では，タスクの間に重複を作らない**．
  - 分割したデータの間に重複ができないように分割する．
  - 分割した計算の間に重複ができないように分割する．

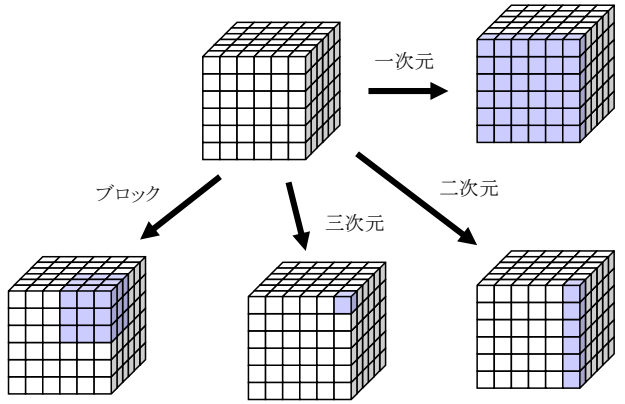
14

領域分割

- 処理すべきデータを，できるだけ小さな単位に分割する．
  - もし可能なら，**なるべく同じような大きさのデータに**分割する．
  - 次に，分割されたデータに適用すべき計算処理をはっきりさせる．
- 分割するデータの候補
  - プログラムに与える**入力データ**
  - プログラムによって得られる**出力データ**
  - 実行途中で保持しておかなくてはならない**中間データ**
- 上記の内で最も大きなものを最初に考えるようにすると，当たっていることが多いだろう．

15

領域分割の例



16

機能分割

- 実行すべきすべての計算を，より小さなまとまりに分割する．
  - 分割法の候補
    - 大量の計算を，小さなタスクに分割する．
      - 例：チェスプログラム
        （「次の一手」，「その次の一手」，…がたくさんあるので，何手か先の可能な一手のそれぞれをタスクとする．）
    - 解こうとする対象が異なる振る舞いをする「もの」の集まりになっている場合，それぞれの「もの」をタスクとする．
      - 例：全地球規模の気候シミュレーション
        （大気，海洋，河川，両極の氷，二酸化炭素発生源，etc.）
      - この場合，あまりたくさんのタスクを考えることはできないので，それぞれのタスクをさらに領域分割することになりそう．

17

分割設計に関するチェックリスト

- タスクの数は，実行計算機のプロセッサ数より少なくとも1桁以上大きくなっているか？
- タスクの間に冗長な計算や冗長なデータ配分は含まれていないか？
- タスクの「大きさ」はほぼ同じになっているか？
- 解こうとする問題の規模が大きくなるとタスクの数も増えるようになっているか？
- 他の可能な分割法は検討したか？

常にすべてYESになるようにできるとは限らないが，可能な限り，できるだけ多くYESになるようにするのがよい．

18

2．通信設計

- 分割段階で得られたタスクが完全に独立に実行できることはほとんどない．
  - 計算が先に進むために，他のタスクの情報を必要とすることが多い．
- 通信構造を明確化する際のおおまかな手順
  - どのタスクがどのタスクの情報を必要とするか？
  - そのときに受け渡しされるのはどのような情報か？
- 分割設計段階での難易度とは逆に…
  - 機能分割されたタスクの間の通信構造は，データの流れとして自然にとらえられることも多い．
  - 領域分割されたタスクの間の通信構造は自明ではないことが多い．

19

局所的通信と大域的通信

- **局所的通信**
  - 各タスクが，ごく少数のタスク（隣人，neighbor）とだけ通信すればよい．
  - この場合は，それぞれのタスクの行うべき通信を，送り手対受け手が1対1となる send/receive とする．
- **大域的通信**
  - 各タスクが，多くのタスクと通信しなければならない．
    - それは1対1のsend/receive の集まりで代用できるか？
    - だめなら，1対1のsend/receive の代わりに，以下のような通信形態が利用できないか，検討する．
      - broadcast
        - ある1つのタスクが持っている情報を，他のすべてのタスクに送信する．
      - reduction
        - ある1つのタスクに，他のすべてのタスクが持っている情報を集めてくる．

20

構造化通信と非構造化通信

- **構造化通信**(structured communication)
  - 通信のトポロジが規則的なパターンに従っている.
    - 例:上下左右, 木構造
  - send/receive を用いる場合, 通信相手の指定が比較的簡単.
  - 次の凝集化段階で, ほぼ同じ大きさのタスクにまとめていくのが, 比較的容易.
- **非構造化通信**(unstructured communication)
  - 通信のトポロジに規則性がない.
  - send/receive を用いる場合, 通信相手の指定がやや面倒.
  - 次の凝集化段階で, ほぼ同じ大きさのタスクにまとめていくのは, やや難しくなる.

21

静的通信と動的通信

- **静的通信**(static communication)
  - 通信のトポロジが, プログラム実行の途中で変化しない.
    - send/receive を用いる場合, 通信相手の指定が比較的簡単.
    - 次の凝集化段階で, ほぼ同じ大きさのタスクにまとめていくのが, 比較的容易.
- **動的通信**(dynamic communication)
  - 通信のトポロジが, プログラム実行の途中で変化していく.
    - send/receive を用いる場合, 通信相手の指定がやや面倒.
    - 次の凝集化段階で, ほぼ同じ大きさのタスクにまとめていくのは, やや難しくなる.

22

同期通信と非同期通信

- **同期通信**(synchronous communication)
  - 通信が完了するまで, 送り手も受け手も, その次の処理に進めない場合に用いる.
- **非同期通信**(asynchronous communication)
  - 受信が終了する前に送り手が次の処理に進んでもよい, または, 受け手の準備状況に関わらず送り手が送信データを次々に用意することができる場合に用いるとよい.
    - 送り手と受け手の「待ち合わせ」の間に, 他の計算を並行して進めることができるようになる.
    - ただし, 送信が完了する前に受け手が受信の次の処理に進むような通信形態は誤動作の原因になることが多いので, 普通はあまり考えなくてよい.
  - 受信の準備ができた, あるいは, 受信の必要が出てきた時点で, 受け手がデータを要求することが必要になる.
    - 「データをくれ」,「データはないか?」といったメッセージをやりとりする.
    - send/receive の代わりに,「受け手がデータを取りに行く」という操作を考える.

23

通信設計に関するチェックリスト

- 各タスクは, ほぼ同じ数のメッセージをやりとりするか?
- 各タスクが通信しなくてはならない相手の数は少ないか?
  - もしそうでない場合には, 局所的通信で代用できるか, 大域的通信が必要になるか, 考える.
- 通信処理は, 他の処理と並行に進められるか?
- 異なるタスクの間の通信は並行に進められるか?

常にすべてYESになるようにできるとは限らないが, 可能な限り, できるだけ多くYESになるようにするのがよい.

24

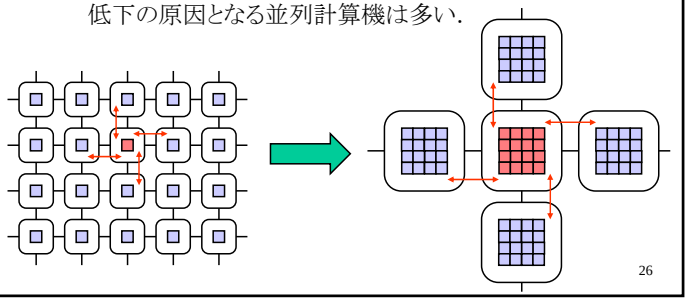
3. 凝集化設計

- 分割設計段階と通信設計段階までは:
  - 実行計算機の構成・機能は考慮していない.
    - 計算機の性質によって，効率的に実行できたりできなかったりする.
  - タスク間にデータの重複・計算の重複はない.
- 凝集化設計段階では，以下のような点に留意しながら，**タスクをより大きな単位へとまとめていく**.
  - 予想される実行性能
    - タスク間の処理時間のばらつき・通信の偏りを避ける.
  - タスク間にデータまたは計算の重複を作ることによる性能向上の可能性
- 上記のような検討の結果，プロセッサ数と同じ数のタスクにまでまとめることができる場合
  - プロセッサへの割り当ては自明.
  - MPI型のSPMD実行可能なプログラムを作るのも，さほど困難ではない.

25

粒度の拡張

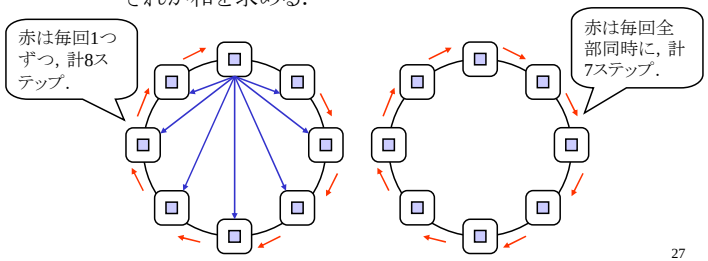
- 領域分割で生まれたいくつかのタスクを一つにまとめると…
  - 「**面積-体積効果**」
    - 分割数が減ると，特定のタスクとタスクの間で1回にやりとりするデータは増えるが，メッセージの総数は減少する.
    - 1メッセージ当たりの大きさよりもメッセージの多さが性能低下の原因となる並列計算機は多い.



26

凝集化設計段階でのその他の検討事項の例

- 通信・計算の重複実行
  - すべてのデータの総和を，すべてのプロセッサで求める
    - リング状の通信パターンで総和を求めてから，結果を他のプロセッサに broadcast する.
    - リング状の通信パターンで，データを順送りしながらそれぞれが和を求める.



27

凝集化設計に関するチェックリスト

- 凝集化により，通信のコストを下げられるか？
- 凝集化により計算の重複が生じる場合，それに必要なコストがそれによる性能向上のメリットを上回っているか？
- 凝集化によりデータの重複が生じる場合，それによってアルゴリズムの規模適応性が損なわれることはないか？
- 凝集化の後でも，各タスクの通信コスト・計算コストは，ほぼ同じになっているか？
- 凝集化の後でも，解こうとする問題の規模が大きくなるとタスクの数も増えるようになっているか？
- 凝集化により並行性がやや失われたとしても，並行に実行できる部分は残っているか？
- 負荷の不均衡・プログラム作成時の多大な負担・規模適応性の喪失といったことを起こさずに，さらにタスク数を減らすことはできるか？
- 逐次型プログラムを並列版に書き換える際に予想される手間，できあがったプログラムの複雑さは我慢できる範囲にあるか？

28

4. 割り付け設計

- タスクをプロセッサへ割り当てる際の2つの基本方針
  - 矛盾することもある.
    - 並行に実行できるタスクは，別のプロセッサに割り当てる.
    - ひんぱんに通信するタスク群は，同じプロセッサに割り当てる.
  - 全体の処理コストが小さくなるように，妥協点を見い出す.
- 同じような「大きさ」のタスク間に規則的な通信構造がある場合
  - 通信のコストだけを考えて割り当ててよい.
- タスクの「大きさ」が一様でない，または，通信構造が規則的でない場合
  - 各プロセッサが受け持つタスク群の処理負荷の和が均衡するように割り当てる.
- タスクの数，各タスクの計算量・通信量が実行中に変化する場合
  - 実行中に負荷を均衡化させることを考える.
    （実行中にタスクのプロセッサへの割り当てを変更していく.）

29

負荷の均衡化

- 主に，領域分割を採用する場合に用いる.
  - 最初の領域分割後のタスク（または凝集化後のタスク）の設計を少しだけ「軌道修正」する.
    - 負荷が大きくなりそうなタスクがあれば，境界付近のデータを隣のタスクに移して，その後の段階の設計をやり直す.
      - これがうまく行けば，SPMD実行は簡単.
    - 凝集化設計段階でタスク数をあまり減らさず，たくさんのタスクをプロセッサにランダムに割り当てる.
  - 実行の途中で，最初の領域分割後のタスク（または凝集化後のタスク）を少しだけ「軌道修正」して，実行を継続する.
    - 周囲よりも負荷が大きいタスクが出てきたら，境界付近のデータを隣のタスクに移して，実行を継続する.

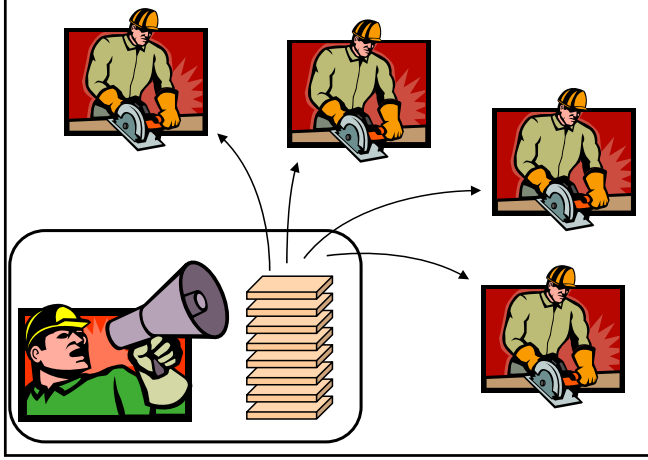
30

タスクスケジューリング

- 主に，機能分割を採用する場合に用いる.
  - **管理者・労働者型**（集中管理型）
    - manager が1個，worker が（プロセッサ数－1）個
    - worker が行うべき「処理」を小さく分割（その総数がプロセッサよりも十分多くなるように）しておく.
    - 分割された「処理」は，すべて manager の管理下に置いておく.
    - worker は，自分に割り当てられた「処理」が終わったら，次に実行すべき「処理」を manager のところに取りに行く.
  - **非集中型**（decentralized scheme）
    - プロセッサ数と同じ数の worker だけ
    - worker が行うべき「処理」を小さく分割し，各 worker に分配してから処理を始める.
    - ヒマになった worker は，隣近所の worker から「処理」を分けてもらう.
- タスクスケジューリングの問題点
  - 全体の処理が終わったことをうまく判定するしかけを組み込む必要がある.

31

Manager/Worker



32

割り付け設計に関するチェックリスト

- 複雑な問題をSPMD実行しようとする場合，動的にタスクを生成・消滅させるような代替手段も考慮したか？
- 動的なタスク生成・消滅にもとづく設計をした場合，SPMD実行させるような代替手段も考慮したか？
- ランダムなプロセッサ割り当てを行う場合，少なくともプロセッサ数の十倍以上のタスクがあるか？
- 実行時に負荷の均衡化を行う場合，負荷情報の採取・負荷の移動・プログラム作成の難易度が異なる複数の方式を比較検討したか？
- manager/worker型のタスクスケジューリングを採用する場合，managerがボトルネックにならないことは確認できているか？

33