マイグレーションを支援する分散集合オブジェクト

高橋 慧　田浦健次朗　近山 隆 （東京大学）

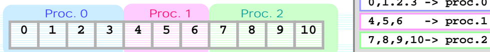
背景・目的

グリッドプログラミングにおける要請

- 動的なプロセッサ数の変化に対応させたい
  - ▶使用可能なプロセッサが動的に変化
  - ▶利用可能な資源を最大限活用
- 記述を簡単にしたい
  - ▶オブジェクト指向
- 無駄な通信が発生しない

対象とする集合

- インデックスで要素を識別できる集合
  - ▶配列・ハッシュ表…
- 分散環境では断片に分割して保持
  - ▶プロセッサが固定
    →要素の位置を簡単に計算できる
  - ▶プロセッサ数が動的に変化
    →どの要素がどのプロセッサにあるか不明
    →要素とプロセッサの対応表が必要



```
0,1,2,3 -> proc.0
4,5,6   -> proc.1
7,8,9,10-> proc.2
```

既存モデルでの問題点

- メッセージパッシング
  - ▶送信先の指定にプロセッサ番号を用いるため、プロセッサ増減に対応が難しい
  - ▶対応表を一から実装する必要がある
  - ▶手続き的で記述が難しい
- 分散共有メモリ
  - ▶プロセッサ増減自体には容易に対応
  - ▶データと仕事の対応を意識しないと、不必要なリモートメモリアクセスが多発し、低速
- 分散オブジェクト
  - ▶オブジェクト単位ではマイグレーション可能
  - ▶プロセッサ間にまたがるデータは簡単には扱えない

提案

「分散集合オブジェクトモデル」

- プロセッサ間にまたがる集合を一オブジェクトに
  - ▶プログラマは断片オブジェクトを定義
  - ▶メソッドは引数にインデックスの範囲を取る
  - ▶オブジェクトの外側からは、集合全体に対しメソッドを呼び出す
  - ▶断片は分割・併合・マイグレートできる

メソッド呼び出しの記述

断片の定義

- ▶集合全体の一部を持つものとして定義
- ▶インデックス集合とデータを保持
- ▶メソッドは引数に「インデックス集合」を取る

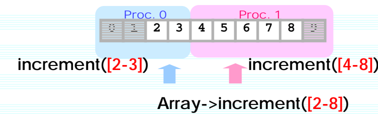
メソッド呼び出し(RMI)

- ▶インデックス集合とメソッドを指定
- ▶各断片が持つインデックス集合と重なりを取って、メソッドが呼び出される

例:分散配列の要素をインクリメント

- ▶配列によらず要素[2-8)をインクリメント

```
class DistributedArray {
  increment(IndexSet *is){
    foreach (index <- is){
      data[index]++;
    }
  }
};
int main(){
  WholeArray->increment ([2-8]);
```



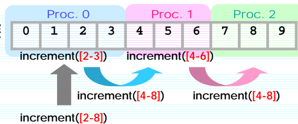
▶データアクセスにプロセッサ番号を用いないため、プログラマは配分を意識せず記述できる

▶メソッドはデータのあるプロセッサで実行されるため、データに応じてタスクも自動的に配分される

処理系の実装

- ▶要素とプロセッサの対応を集中管理すると…
  - 実装は楽
  - 対応表を持つプロセッサにメッセージが集中
- ▶Phoenixライブラリを利用 (http://www.logos.ic.i.u-tokyo.ac.jp/phoenix/)

  メッセージパッシングライブラリの一種。各プロセッサは番号の集合([0-100)など)を持ち、メッセージは一つの番号に向けて送信される。番号とプロセッサの対応はプログラマが自由に変更でき、この対応は各プロセッサが分散保持している。
  - インデックス一つをPhoenixの番号一つに対応付け
  - 要素とプロセッサの対応が分散保持される
  - メッセージが対応表を持つプロセッサに集中しない
- ▶メソッド呼び出しメッセージの中継方法
  - 呼出先の先頭のインデックスにメッセージを送信
  - その断片のインデックス集合と重なりを取って、残りを次の先頭インデックスに送信
  - ツリー状の中継も実装



マイグレーションの記述

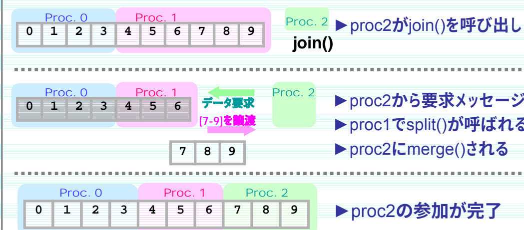
断片の定義

- ▶断片の定義で、split()・merge()を記述
- ▶分割・併合に必要な処理を記述

計算に参加 / 脱退

- ▶プログラムは外部からjoin() / leave()を呼び出し

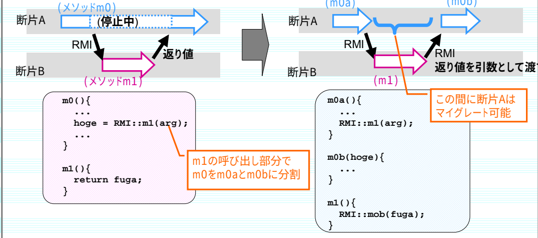
```
class DistributedArray{
  split(void){
    (データとインデックス集合を分割)
    return (分割したデータ);}
  merge(データ){(データを併合)}
};
int main(){
  join(object_id);
```



▶プログラマはクラス定義でsplit(), merge() を記述すれば、関数1つでマイグレーションが実現される

マイグレーションとスレッド

- ▶通常の処理系では、実行中のスレッドは移送できない
- ▶本モデルではメソッド実行中の停止を禁止することで、任意のメソッド実行の間でマイグレーションを可能にする
- ▶あるメソッドの返り値を必要とする処理は、そのメソッドの呼び出し前と呼出し後に分割して記述する
- ▶一断片内での実行スレッドは一つに制限でき、このスレッドが待機状態であればマイグレーションが可能である
- ▶将来的には、プリプロセッサにより、擬似的に返り値を取れるような処理系を構築したい
  (スタックの情報はRMIの引数の形で授受する)




```
m0(){
  ...
  hoge = RMI::m1(arg);
  ...
}
m1(){
  return fuga;
}
```

```
m0a(){
  ...
  RMI::m1(arg);
}
m0b(hoge){
  ...
}
m1(){
  RMI::m0b(fuga);
}
```

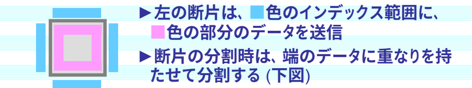
マイグレーション対応SOR

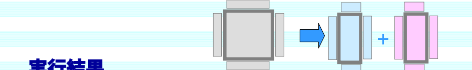
基本的なアルゴリズム

- ▶隣接する境界のデータを交換
- ▶データを更新
- ▶残差を一カ所に集計

分散集合オブジェクトを用いた記述

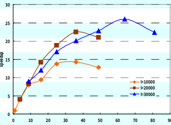
- ▶各断片がメソッドを呼び合う形で処理が進行
- ▶境界データの交換
  - 端のデータを引数に、隣のインデックスに対しRMI
  - データを必要とする要素のインデックスに向けて送信
  - 断片が移動しても、正しい断片にメッセージが届く
- ▶右のコードにおいて、アルゴリズムの正しさを証明した





実行結果

- ▶プロセッサ81台の環境で正しく動作
- ▶64台で連続joinに成功
- Mflops値
  - ▶一回のループ時間から計算
  - ▶一辺10000, 20000, 30000で実行
- 台数増加により頭打ち
  - ▶現状の実装では処理系のオーバーヘッドが大きいためと推定



まとめ

- 分散集合オブジェクトモデルを提案・実装した
  - ▶インデックス集合を指定したRMIにより、プロセッサ数増減に対応
  - ▶オブジェクト指向による記述
  - ▶対応表を分散保持しメッセージが集中しない
- これを用いてSOR法のプログラムを記述した
  - ▶正しく動作することを確認した
  - ▶アルゴリズムの正しさを示した

今後の課題

- 実装の改良による処理系の高速化
- 他のアプリケーションの記述
- 記述性の改善
  - ▶返り値についての検討

powered by Phoenix Library