今後のHPC技術の研究開発課題とロードマップの検討状況ならびに意見交換 -- プログラミング言語・モデル –

執筆者： 丸山(東工大)、滝沢（東北大）、田浦（東大）、平石（京大）、窪田（広大）、八杉（京大）、中尾（筑波大）

アドバイザ： 佐藤（筑波大）、中島（京大）、米澤（理研）

概要

プログラミング→アーキテクチャとアプリケーションをつなぐインターフェイス

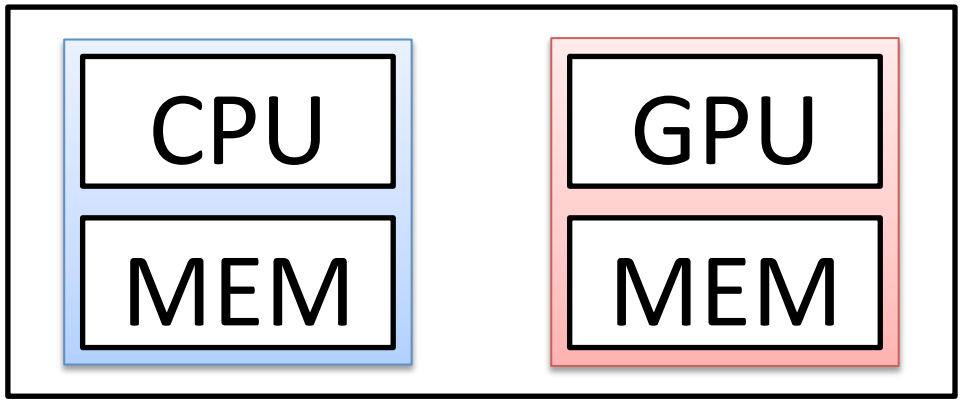
今後特に重要になる問題

- ヘテロジニアスアーキテクチャ
- メモリ階層・モデル
- 大規模並列
- 電力
- 耐故障性

性能と生産性の両立

それらを解決するためにやるべき研究開発とは？

ヘテロジニアスアーキテクチャ



- 新規プログラム開発
- アルゴリズムのヘテロジニアスアーキテクチャへの最適化
  - 問題の分割と割り当て
- 可搬性
  - 標準プログラミング言語・ライブラリの欠如
  - 性能可搬性の欠如

ヘテロジニアスアーキテクチャ

- 現在の対策・現状
  - 標準プログラミングモデル・環境の研究開発
    - 現在はハードウェアを強く意識したプログラミングモデル
    - 性能最適化は共通化されていない
- 今後のアプローチ
  - 2012～2017: インタフェースの共通化、標準プログラミング言語からの利用
    - 半自動最適化や自動チューニング技術
    - アクセラレータとのデータ転送の自動化・高効率化
    - 複数種類のアクセラレータへの対応 (e.g. dGPUとAPU)
    - 新機能の有効活用方法の検討（アクセラレータ間通信など)
  - 2018～2020: 消費電力などの制約からヘテロ化がさらに進行
    - ヘテロな環境を使いこなす技術がますます重要
    - 現在とは全く異なるアーキテクチャのアクセラレータ
      - 例えばFPGAのような再構成可能ハードウェア

メモリ

- 課題：複雑化するメモリアーキテクチャ
  - 現在：分散メモリシステム，GPGPU → MPI, CUDAで苦労して書いている
  - 今後：さらなる大規模分散化，ノード内でもキャッシュ階層の複雑化，電源の一部動的停止，不揮発メモリ採用
  - 今後，さらにアーキテクチャを意識したプログラミングが求められ，生産性低下が対応できないほど深刻になる可能性
- 過去・現状の取り組み
  - 完全自動化（e.g., HPF）は過去に失敗
  - アーキテクチャを意識させつつ，うまく抽象化することで記述を簡便化させるものが現在の主流（PGAS，ディレクティブベース）
  - ノード内のメモリ複雑化に対応したものはまだ少ない（Sequoia++など）
- 今後のアプローチ
  - ～2017
    - 現在開発中のPGAS／ヘテロ対応言語の普及（実装の洗練，ライブラリの充実等）
    - ノード内複雑化対応に向けた開発．たとえば，キャッシュのソフトウェア管理，ローカルストア，不揮発メモリ，電力停止などをどう抽象化するかの検討，プロトタイプ実装
  - ～2020：ノード内複雑化対応言語の普及

大規模並列システムでの Strong Scaling

- バンド幅向上のための更なる階層化
  - 多階層キャッシュ、ネットワークの階層化
- 相対的なレイテンシ増大による速度向上の限界

→ 階層化されたシステムでのレイテンシ削減

  - 低レイテンシネットワーク構成、通信機能高度化、階層のバイパス

→ 大規模並列性・階層されたシステムを有効に活用できるソフトウェア

大規模並列

- 課題
  - 階層的かつ膨大な並列性
    - 非常に多数のノード，ノード内の多数のコア，コア内の多数のSIMD並列性
    - アクセラレータ活用の必要性
  - NUMA構成・非均質ネットワーク
    - 構成を意識したプログラミングが必要
- 現在の対策・現状
  - 階層的な並列性を利用するために各階層用に設計されたプログラミングモデル
    - 統一的な記述から各階層の並列性を利用する必要性
  - 大規模並列用のアルゴリズム再設計
    - 現在主流のSPMD方式では素直に表現できない等の問題
- 今後のアプローチ
  - 階層的並列性の統一的利用
  - 大規模並列用に設計されたアルゴリズムのサポート

電力

- 課題
  - 消費電力制約
  - ハードウェア・ソフトウェアの協調による電力効率最大化
- 現在の対策・現状
  - SWからみたHWの消費電力/消費エネルギーに関する標準インタフェースすら整備されているとは言い難い
- 今後のアプローチ
  - HW/SW間標準インタフェースが整備され，それを利用して，システムソフトウェアが主体(or管理者)となり電力制約下での性能最大化

耐故障性

- 耐故障コスト増大&故障率の増加への対応限界
  - 主として要素数、規模の増加が原因
    例：チェックポイントを共有ストレージへ保存する時間がシステムのMTBFを越える
    - ペタスケールマシンで数十分かかるケースも
- 横断／連鎖的故障の複雑化
  - システムの複雑性により故障要因が判明しづらい
  - 故障の局所化が出来ず全スタック復旧になりがち
- 難検知故障の増加
  - ECC の対応限界を越えたデータ化け
  - 故障検知能力を備えない要素の増大
    - 通信ケーブルやCPUレジスタ等

耐故障性

- 課題
  - エクサスケールでは耐故障性を考慮がジョブの実行に不可欠(MTBFが短くなるため)
  - アプリケーションから、故障の検知/復旧を制御する必要
- 現在の対策・現状
  - システム全体のチェックポインティング/リスタート
  - MPIの耐故障機能の標準化
- 今後のアプローチ
  - 言語レベルでチェックポインティング・復旧方法を指定し、アプリケーションプログラマの耐故障記述の負担軽減
  - Master-Worker型など耐故障性を記述しやすいプログラミングモデルのサポート
  - 故障予測精度の向上を前提とした、言語レベルでの故障への事前対処記述

想定プログラミング環境

- これまでの基礎研究を基盤とした将来のあるべきプログラミング環境とは？
  - 「性能が出て、生産性があり、どんなシステム環境でも使え、柔軟性がある」は並立困難なのでどうするべきか

- 想定環境の例
  1. 既存アプリとの親和性重視のアプローチ
  2. ドメイン特化による高水準化のアプローチ

アプリケーション移行支援

- これまでに蓄積されてきた膨大なソフトウェア資産の継承
  - そのままの状態ではエクサシステムで高性能を達成できない
  → なんらかの手段でアプリケーションを進化させる必要性
- やるべき研究開発
  - 移行コストの軽減手法の研究
    - 既存コードをいつまで使い続けられるか？/続けるべきか？
      - アプリケーション基本構造の再設計・再実装コストと移行コストの比較
    - ランタイムシステムでどこまで透過的に隠ぺいできるか？/すべきか？
      - かしこい実行時環境が現システムと新システムの差を隠ぺい
    - 移行作業をどこまで自動化できるか？/すべきか?
      - XcalableMPやOpenACCなどのディレクティブベースは親和性高
  - エクサ時代のアプリ設計と既存コードの再利用性
    - アプリを基本構造から再設計？設計指針はまだわからない

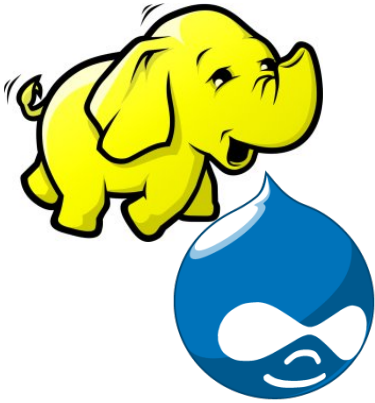
段階的に移行
（心理的障壁低）

将来のアーキテクチャの変化
への継続的な適応作業

再実装コスト >> 移植コスト＋支援環境開発コスト

- なにを支援すべきか/できるのか？
  - コードの設計、可読性、保守性、可搬性の経年劣化対応
  - 適切な実装・アルゴリズムの変化対応
- 方針案
  - コード修正のガイドラインの作成
    - 典型的な計算パターンごとに方針を明確化
    - ガイドラインに沿ったコード修正を支援するツール整備、など
  - 実装/システム依存性を軽減する抽象化
    - システム構成に依存した実装の回避/隠ぺい
    - アルゴリズムやデータ構造の詳細を隠ぺい、など
- 推奨ロードマップ
  - 計算科学者と計算機科学者の合意によるガイドライン作成とそれに基づく支援ツール開発
  - 成功事例の提示とガイドラインへのフィードバック

アプリケーションフレームワーク



- 特定ドメインに特化したプログラム開発環境
  - 例： Ruby on Rails、Drupal、Hadoop、SQL
  - 生産性、性能、可搬性を実現、しかし汎用性は欠如
- 計算科学向けフレームワーク
  - アプリケーションドメインのボキャブラリでプログラムを記述
  - 計算機のボキャブラリでは記述しない
  - 例1： OpenFOAM
    - 流体向けフレームワーク。離散化、データ入出力、など共通計算パターンを抽出しライブラリとして提供
  - 例2： Listz
    - 非構造計算向けフレームワーク。各種アーキテクチャ向け実行コードを自動生成

一度フレームワークへの移植が必要（初期コスト大）

将来のアーキテクチャの変化への容易な（自動的な）適応

フレームワーク開発コスト
+
フレームワーク上アプリ開発コスト
<< アプリ開発コスト

- 方針案1: 開発フレームワークの選別
  - アプリケーションの計算パターンの特定
    - Berkeley motifs、Mini apps など
  - 計算パターン毎にフレームワークを開発
    - 個々のパターンを抽象的に記述可能なプログラミングインターフェイス
    - 計算パターンに特化した並列化、負荷分散、など
- 推奨ロードマップ
  - 現状はフレームワークを種々のドメインに対して開発し知見を積む段階
    - 種々のアーキテクチャ向け最適化フレームワークを開発
  - アプリケーション開発者との共同作業
  - フレームワークの柔軟性と個々のアプリへの適用性のトレードオフを調査

フレームワーク開発コスト
+
フレームワーク上アプリ開発コスト
<< アプリ開発コスト

- 方針案2: フレームワーク開発コストの削減
  - (ちょっと計算機に強い)計算科学者でも作れるように
  - フレームワーク構成コンポーネントを部品として用意→部品を組みあわせることでアプリケーションドメインに適した設計のフレームワークを構成
    - 負荷分散スケジューラ、アクセラレータ対応、チェックポインタ、など
  - 概して一般化を進めると性能は犠牲→性能との両立が課題
- 推奨ロードマップ
  - まずは単体ドメイン向けフレームワークの開発を通じて、フレームワークとして持つべき機能の洗い出し
  - フレームワークとしての汎用性と性能を両立のための開発方法論の確立

全体構成

- 概要:エクサスケールにおけるプログラミング言語および環境
  - 性能と生産性の両立
    - 現在のヘテロシステムですら生産性の低下は大問題
    - 電力効率，耐故障性，大規模並列，深いメモリ階層などの課題を克服しつつ両立する必要
- 要素技術研究項目
  - ヘテロジニアスアーキテクチャ (滝沢)
  - 大規模並列性 (田浦)
  - メモリ (平石)
  - 耐故障 (窪田)
  - 電力 (八杉)
  - 生産性・ツール(中尾)
- 想定プログラミング環境（丸山、滝沢)
- まとめ
  - 開発体制や普及戦略などに言及

開発体制および普及活動

- 開発体制の確立
  - 1つの研究室だけではなく、複数の大学や研究所、特に企業と協力した開発体制
- 普及活動
  - 既存の大規模アプリケーションを新言語やフレームワークで開発し、実用的な大規模アプリケーションが開発可能であることを示す
  - （国際的な）コミュニティの形成
    - 標準化活動
    - 他の有用なHPCソフトウェアとの連携
    - メーリングリスト
    - プログラム講習会
    - ワークショプの開催

まとめ

- エクサスケールのシステム構成は複雑化
  - ハードウェア構成を意識する必要性＝生産性低下
  - 加えて電力効率や障害対応などにも配慮する必要
- ロードマップの構成
  - 個々の課題に対する要素技術の検討
  - 要素技術を統合するプログラミング環境の例示
- 開発体制や普及活動