Post-Peta/Supercomputing
-計算機科学から-

米澤 明憲
前田俊行

東京大学大学院
情報理工学系研究科
コンピュータ科学専攻

話の内容

1. SC'10の雰囲気
2. 計算科学と計算機科学の連携・協業
3. これからのアーキテクチャ
4. これからのシステムソフトウエア
5. ポストペタに向けた日本の戦略は？

SC'10とClayton Christensenのメッセージ？

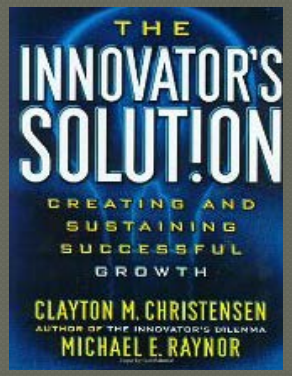
- 「excellent」と言われてきた会社が短い寿命で滅びるmechanismを明らかにした
- high-endの製品にフォーカスしてきた企業は、安価な製品を求めるlow-endの顧客をターゲットとする新企業(disruptive innovations)に、盲点を突かれる

- 技術の進歩速度が、ニーズ増加速度を上回るとそれが破壊的技術(disruptive technology)！
- 技術革新は、顧客の利用可能な範囲を超えて進歩

- sustained成長 対disruptive成長
  - ・大手企業対新興・企業会社
  - ・大仕掛け対お手軽
    - •SUN-Intel-IBM(-Fujitsu)対NVIDIA-ATI
    - • Toyota対Tata
    - •京-BlueWaters対天河(-Tsubame2)

深読み

妄想



SC'10の勝者は？

- SC'10での勝者はGPGPU！
  - GPGPUに関係する講演が多く、CPUとの性能比較がfairでないとの不満が会場で多く出た.
    しかしSC10の勝者はGPGPUと言われていた.
- 汎用マシンと専用マシンの鬩ぎ合い
  - 所詮GPGPUは汎用にはならない！
  - GPGPUはフロップスは稼げる
  - GPGPUはon-chipのmemoryが少ない、
    それを多くするとコストがかかる.

Top500とGreen500

- Top500 (Linpackで競争)
  - Linpack は演算器が多ければ高数値がでやすい?
  - 近年はアクセラレータで点数稼ぎをする傾向?
- Green500 (Linpack/消費電力 で競争)
  - gPUが貢献すると言われている (演算器の数を消費電力の割に増やせる)
  - ハイブリッドな東工大Tsubame2が2位
  - 一方、Sequoiaの前身(BlueGenes/Q)はアクセラレータなしで1位
    - チップの消費電力を下げる方向で (真面目に?) 研究開発を進める方が基礎技術としては役に立つ
  - 京は5位
- ランキングにどれだけ意味があるのか？
  - 大量のコアを速いネットワーク(e.g., infiniband)で繋げればランキングは上位にできる!
    - ランキングを上げるためだけにいろいろコソコソやれば上げられるが…
- 要は、
インパクトの高い計算が、どれだけ持続的に高性能で可能か

CS（計算機科学）の立場

- これまで
  - スパコンシステムを設計
  - 計算科学を下支え/奉仕（？）
  - 計算機科学の中でのスパコン系の研究の地位は？
- ペタ・ポストペタ時代
  - スパコンシステムを設計
  - 計算科学と協業や協調設計(Co-Design)が必要
    - アプリソフトの記述・チューニング
      - 計算機科学系の人がアプリに進出
        - ミドルウエアやＯＳが分かるアプリの人もいるといいが。。。
    - 新しいアルゴリズムの研究開発が必要
    - OS, runtime系、コンパイラ等についての計算機科学の知見がアプリの人々にも必要なので、強い協業が不可欠

計算科学と計算機科学の協業:
NAMDとCharm++ by K.Schulten & S.Kale

Klaus Schulten
イリノイ大物理学科（理論生物）

SanJay Kale
イリノイ大学計算機科学科
（言語システム）

Reflecting on the beginning of this collaboration: In those days (and to some extent even now) computer scientists tended to work on toy problems (n-queens) or simple kernels extracted from applications. It was unusual for them to work on full-fledged application unless they were just "programmers" for it. Scientists and engineers, on the other hand, couldn't quite see the utility of computer scientists. It was Klaus's recognition of the need that gets credit for this collaboration. (probably along with my intention of working with multiple full-fledged applications as set out in my 1994 position paper [http://charm.cs.uiuc.edu/papers/HpccPositionHPCC94.shtml]. Today, with the challenges of petascale and exascale machines, and heterogeneous architectures loom ahead of us, there is even more need for such collaborations. (S. Kale, Dec. 30, 2010)

NAMDとCharm++

- NAMD – 分子動力学シミュレータ
  - イリノイ大のK.Schulten（理論生物学）
  - 既に全米で40000人のユーザがいる
  - どんなコア数でスケールする → BlueWatersへも
  - Charm++言語システムで構築された
- Charm++ （並列処理のためのC++ ライブラリ・ランタイム）
  - 並列オブジェクト指向[Yonezawa77,85]にもとづくプログラミングモデル
    - 並列オブジェクト = 1つの仮想的なCPU (スレッド) + メモリ
    - 並列オブジェクト間の通信は非同期の message passing
      - 並列オブジェクト間ではメモリは共有されない
        - つまり race condition やデッドロックは生じない
  - 負荷分散機能
    - 計算ノードの負荷に応じて並列オブジェクトを生成するノードを決定したり、並列オブジェクトを他のノードに移動 (migrate) できる
  - MPI インターフェイスもある
    - AMPI (Adaptive MPI): Charm++ の上に実装された MPI
- 想定実行環境
  - 分散メモリ計算環境 (共有メモリ環境でも実行可)

ChaNGa - Charm N-Body Gravityの実装
OpenAtom – 量子レベルのMDの実装、にも。

Schulten came to Kale's office

- Kale　－　1991年頃、メッセージ駆動式の並列オブジェクトベースの言語・実行時システムCharmの開発中、またその頃、流体力学系Ｇと共同研究を始めていた。
- Schulten　－　Kaleのofficeを訪問、Schultenはbiomolecular simulationの研究を説明、彼の研究室にはすでに自作の並列アプリや自作の８ノードのtranspute並列マシンがあった。
  - 彼は、並列計算に経験のある計算機科学者を探していた
- Schulten(PI), Kale, Skeel(二人は計算機科学者)、数値計算の人が、NIHにproposalを書き、1992年にfundingが始まる。
- 1994年までにはNAMDが設計され,実装が検討された
  - (1) ＰＶＭベースのもの　(2) Charmベースのもの,　両方の実装を行った。
  - この二つの実装を比べると、Charmのようなメッセージ駆動のものが対象とするアプリケーションにおいてはるかに優れていることがすぐに判明した。

- 1994年にNAMDはCharm++をベースに再設計・実装
  - マシンはsupercomputers、HP workstationクラスター

- この頃二人のcomputer scientist(数値計算人とn-body問題のコーディングの専門家）が加わった。この共同で、NSFの5年間のgrand challenge fundingを得た。
- NAMDのコードは主として計算機科学の学生とポスドクによって書かれた。生物物理の学生やスタッフも参加。
  - ２０００にはGordon Bell prize のfinalistの一つに
  - ２００２にはGordone Bell Prize（special category)を獲得
- 90年代末には、既成のMDプログラムと厳しい競争
  - NAMDがスケールすることと、
  - 数値計算の結果の正確さで、競争に勝ち
  - unique download数で35000、ユーザは40000人超えている。
  - NIHはNAMDをsupportしてるresource centerに1992年以来今日まで継続的にfundingを行ってきている

1991年当時の米国をを振り返って

- 計算機科学の人が、単なるプログラマーとして働く以外は、full scaleのアプリにのめり込むことは、普通ではありえないことであった。
- 一方、科学者やエンジニア達は計算機科学の(人々の）必要性・有用性を理解していなかった。
- ペタスケール、ヘテロマシンの出現、エクサスケールの時代に向かって、協業の必要性はさらに強まることは明らか！

協業により計算機科学者側が得たもの

- 真の現実問題（NAMD）を解くことで、新しい技法や抽象化・概念(e.g., dynamic load balancing)を得て、Charm++に導入でき、別のアプリに利用に応用出来た。
  - 計算機科学（のプログラミング言語システム系）は、おもちゃのプログラムを動かして評価する傾向にあった
- NAMDの成功が、Charm++に大きな脚光を与えることになった。
  - これだけに脚光をあびるacademiaが作ったスパコンのシステムはほかにない。

BlueWaters (at イリノイ大NCSA) の狙い

- 2012年完成予定？
- 汎用超並列マシン、IBM Power7（+Hubチップ）ベース、水冷方式、…
- 10peta(peak性能), 1peta(sustained性能)
- 資金：DARPA → IBM

- プロジェクト PERCS
  (Productive Easy-to-use Reliable Computing System)

- 現在 30 チームがアプリを開発中
- 開発言語：UPC+MPI
- 開発環境：EclipseをUPC向けに拡張

- フロップ値は問題でない (T. Dunning)：
  - アプリ開発と利用の生産性を高いこと
  - システムやアプリがスケーラブルであること
  - システムやソフトがfault tolerantであること
  - マシンを維持・運営するコストが低いこと

これからのアーキテクチャ

これからのアーキテクチャの傾向

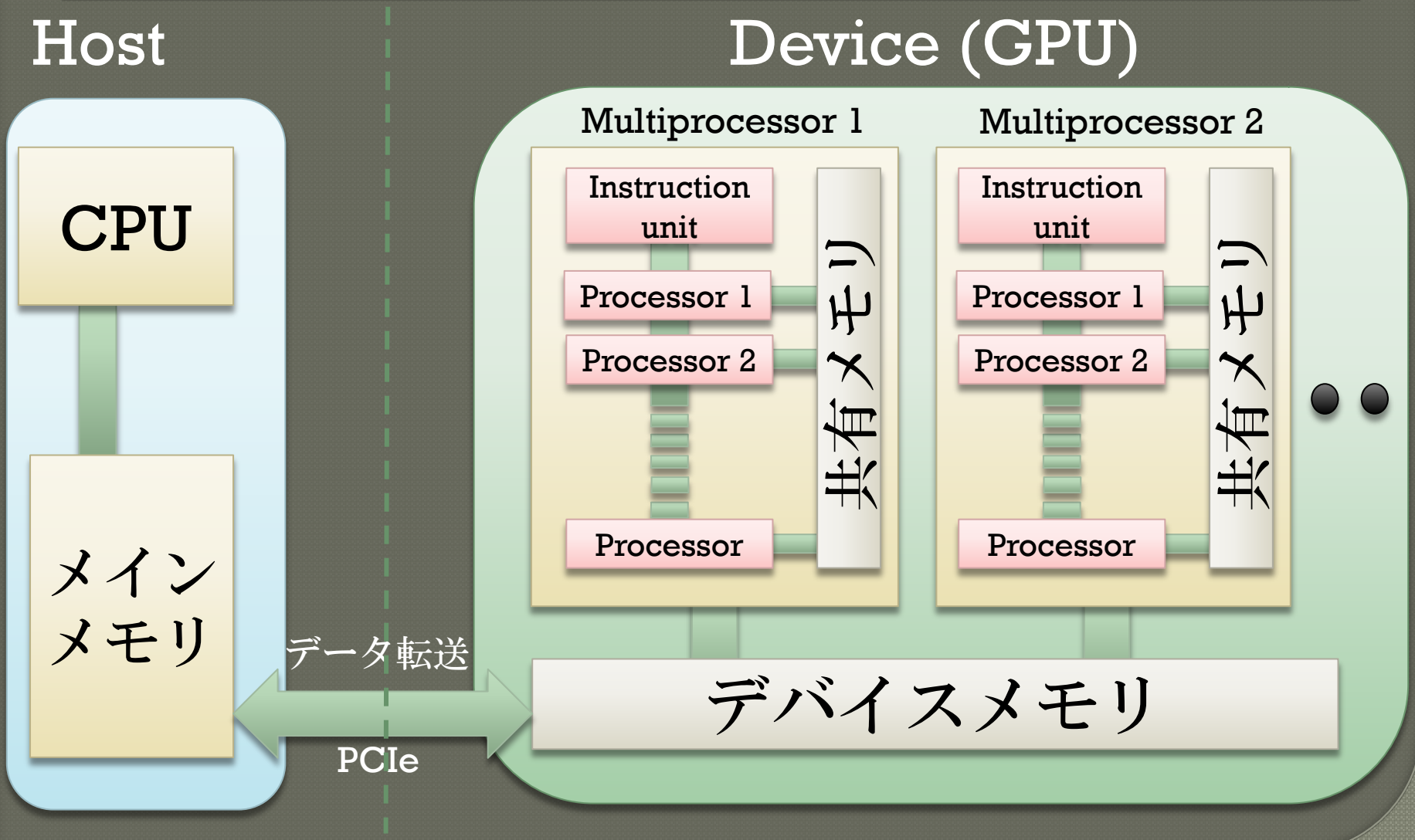
-専用プロセッサと汎用プロセッサ-

既に実用化

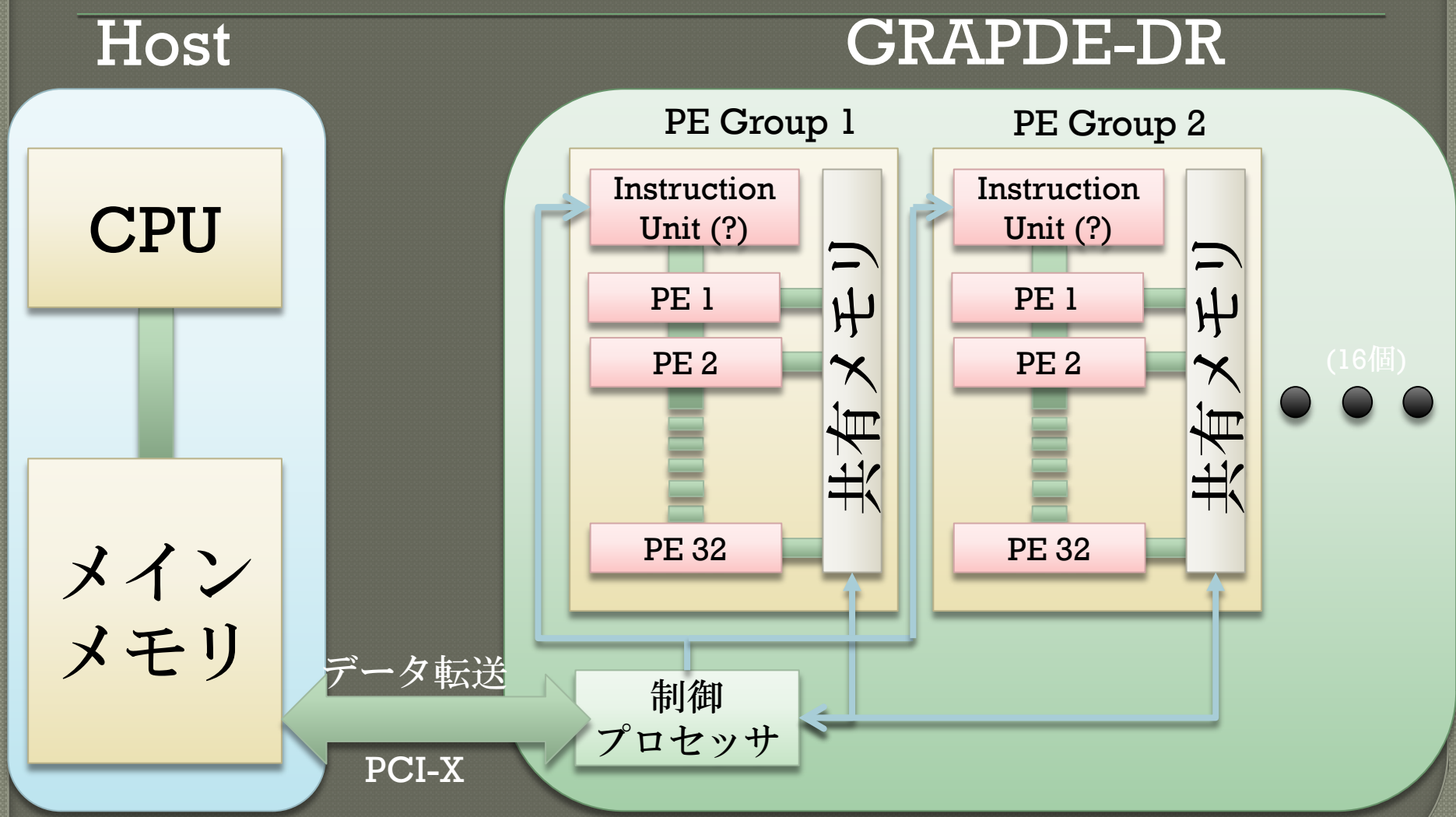
1. 専用プロセッサを加速用デバイスとして汎用CPUに外付けしたノードを多数組み合わせる

2. 専用プロセッサと汎用CPUの一体化（同一チップの中に入れる）傾向が進む！
   - 例えば Intel 社 Sandy Bridge
     (Appleが次期製品に使うという噂,
     SIMD gpu内蔵）2011.1 Intel 発表予定

専用プロのイメージ：NVIDIA Tesla



専用プロのイメージ：GRAPE-DR(牧野)



General Purpose gpu: GPGPU

- 今、なぜGPGPUか？
  - 特定の科学技術アプリにおいてCPUを凌ぐ性能
  - 価格が低い・消費電力が (比較的) 低い
- プログラミングモデル
  - CUDA – Compute Unified Device Architecture
    - NVIDIA社がそのGPU用に用意した
    - ホストCPUからはデバイスとして動く
    - 集中的な計算を「関数(カーネル)」としてGPUにoff-load
    - CPUとGPUのメモリは別々に管理、その間でコピー
    - データ並列・SPMD：GPU上で多数のスレッドが実行する
    - GPUスレッドは軽量で切り替えが高効率

GPGPU の長所・短所

- 利点
  - 異なるデータに対して同一の操作を大量に行うようなプログラムを効率よく実行できる
    - 基本的に GPU は大量の SIMD プロセッサの集まり
    - 液体粒子系シミュレーションで成功（東工大・青木）
      - Navier-Stokes系はＣＰＵに勝てない
  - 安価
- 欠点
  - プログラミングが難しい
    - 広範なアプリでどこまで性能がでるか？
  - GPU の性能を活かすには工夫が必要
    - Host と GPU のメモリ間でデータの転送を行う必要がある
    - GPU 上で高速アクセスできるメモリサイズは大きくない
  - レガシーコードが使えない（変換も困難）
    - 書いたプログラムを長く使えるか？

プログラミングモデルCUDA

- NVIDIA「GPU」向けの C 言語拡張+ライブラリ
- 基本的には SPMD モデル
  - GPU 上に大量のスレッドを割当てて
    コード片 (「kernel」と呼ばれる) を実行する
- ただし以下のような制約がある
  - スレッドのスケジューリングは 32 個単位 (「warp」と呼ばれる)
    - 1 つの warp 内のスレッドは GPU 上で SIMD 実行される
      - つまり 1 つの warp 内で各スレッドが異なる control flow をたどると
        性能が落ちる
  - 高速アクセス可能なスレッド間共有メモリに制限がある
    - 高速共有メモリは「block」と呼ばれるスレッド集合ごとに用意される
      - スレッドは他の block のスレッドの高速共有メモリに直接アクセス不可
      - Block のサイズは最大 512 スレッド
    - 高速共有メモリのサイズは 16 KB
      - しかも同時にアクセスできるのは warp の半分 (= 16 スレッド) まで

ポストペタを指向するアーキテクチャ

汎用超並列マシンの各ノードは4つの可能性：

A．CPU とアクセラレータ GPU (群) を
one chip 化する (e.g., Sandy Bridge, Project Denver)

B．CPU (ホスト) にメニーコアchip (群) を
接続する (e.g., Larrabee)

CPUにはファイル
I/O・タスク管理等を
するOSが搭載可能,
スカラーアルゴリズム

C．CPU とメニーコア (群) を one chip 化する

D．メニーコア (群) のみでホストを構成する
(CPU とメニーコアのコアの区別がなくなる)

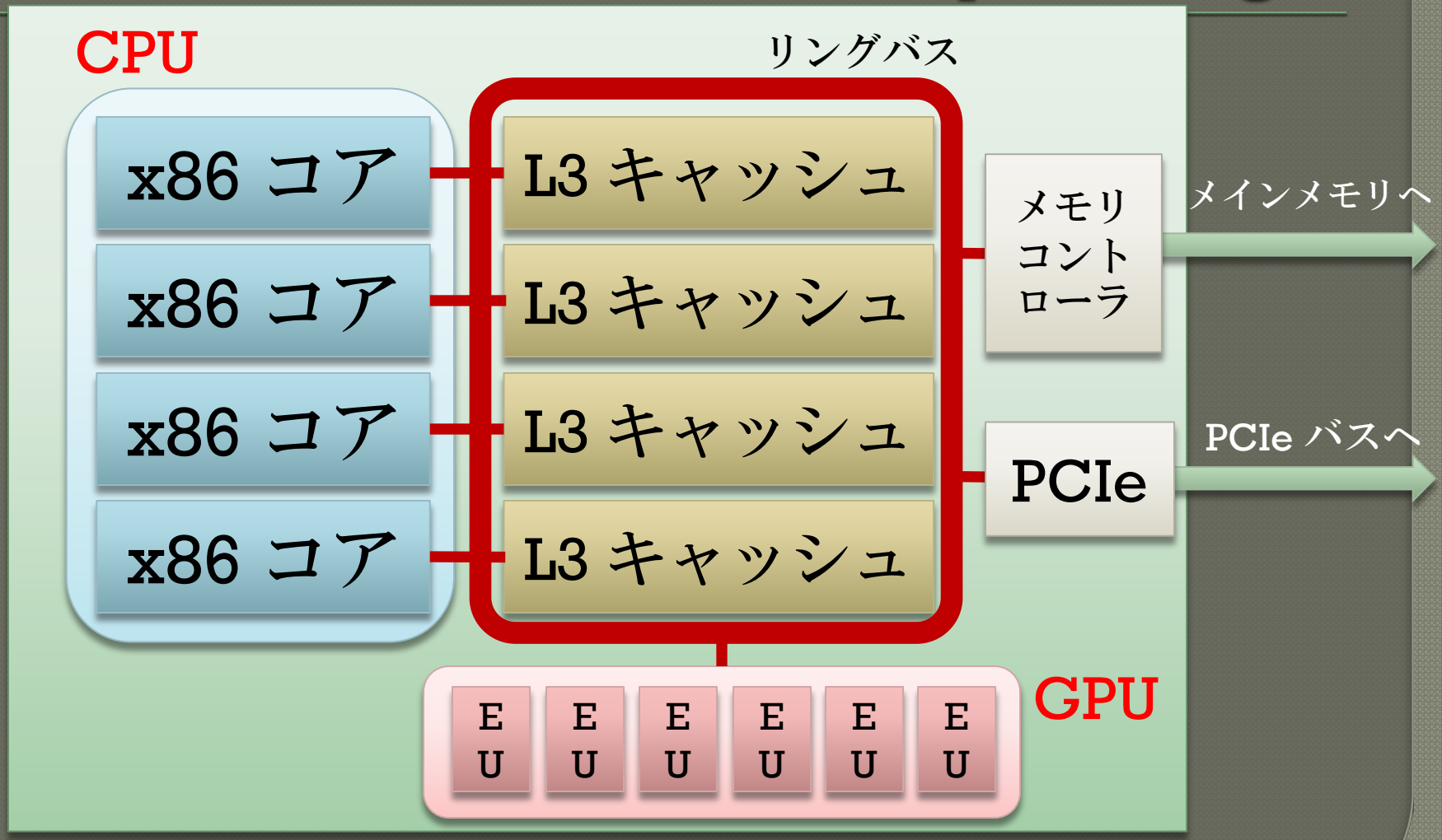
専用マシンシステムの各ノードは：

- 各演算ノードは, 専用計算ユニット群とメモリからなる
- 演算ノード間は高速ネットワークで繋がれる

A Intel Sandy Bridge

- Intel CPU と GPU をワンチップに統合したチップ
  - CPU x86 コア (Westmere) × 4
    - 各コア 2 simultaneous thread サポート
    - 新しい SIMD 演算命令で拡張されている (Intel AVX)
  - GPU コア × 1 または 2
    - 各コア 6 つの EU (Execution Unit) を持つ
  - x86 コアと GPU コアは L3 キャッシュを共有する
    - x86 コア, GPU コア, L3 キャッシュ等はリングバスで接続

A Intel Sandy Bridge



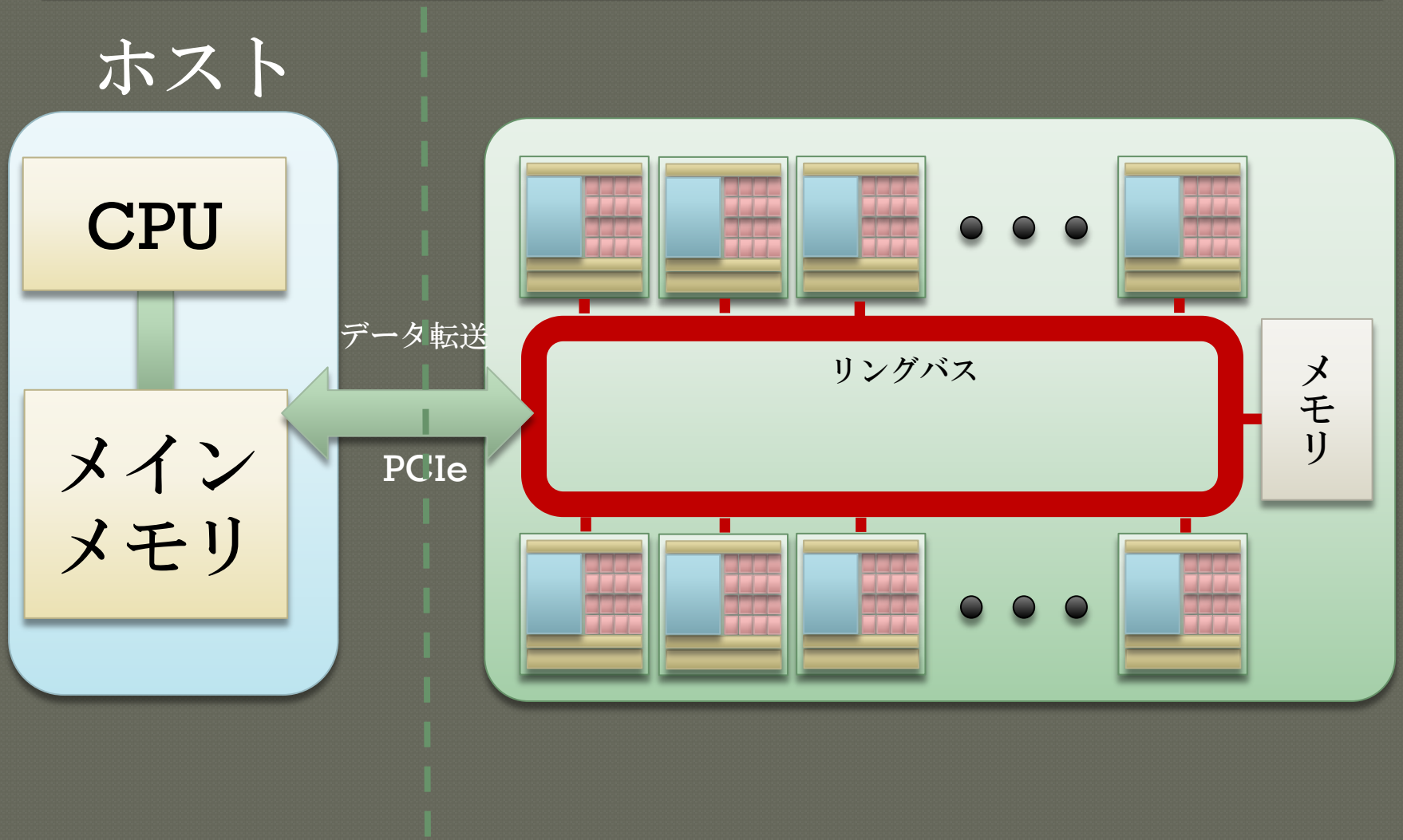
A Nvidia Project Denver

- 2011.01.05発表
  - ARM cpu core とNvidia GPUをワンチップ化??
  - PCからスーパーコンピューターまで広く使う予定

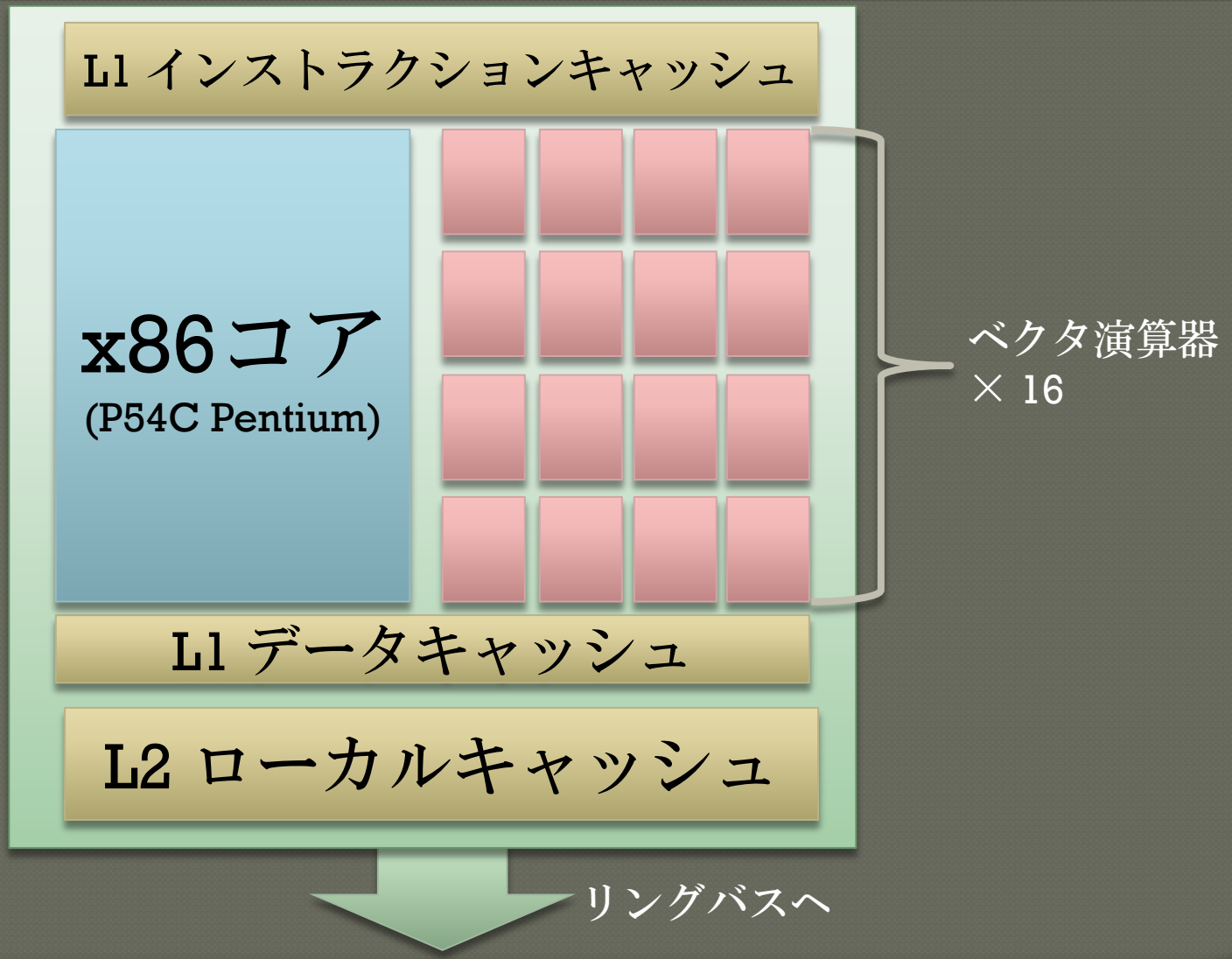
B Intel Many Integrated Core メニーコアチップ("Larrabee")

- 32個 以上 のx86 系 CPU コアからなる
  - 各コアは以下の要素からなる
    - x86 コア (In-order P54C Pentium 1.2 GHz～?) × 1
    - 4 simultaneous thread サポート
      - Larrabee 専用命令で拡張されている
        - 明示的なキャッシュ制御命令など
    - L1 キャッシュ (32KB + 32KB) & L2 キャッシュ (256KB)
    - SIMD 単精度浮動小数点ベクタ演算ユニット × 16
  - 各コアとメモリはリングバスで結合されている
  - キャッシュコヒーレンシ制御あり
- 基本的に x86 バイナリはそのまま実行できる
  - OS でさえ
    - ただし性能を生かすには GPGPU や Cell のようなプログラミングが必要

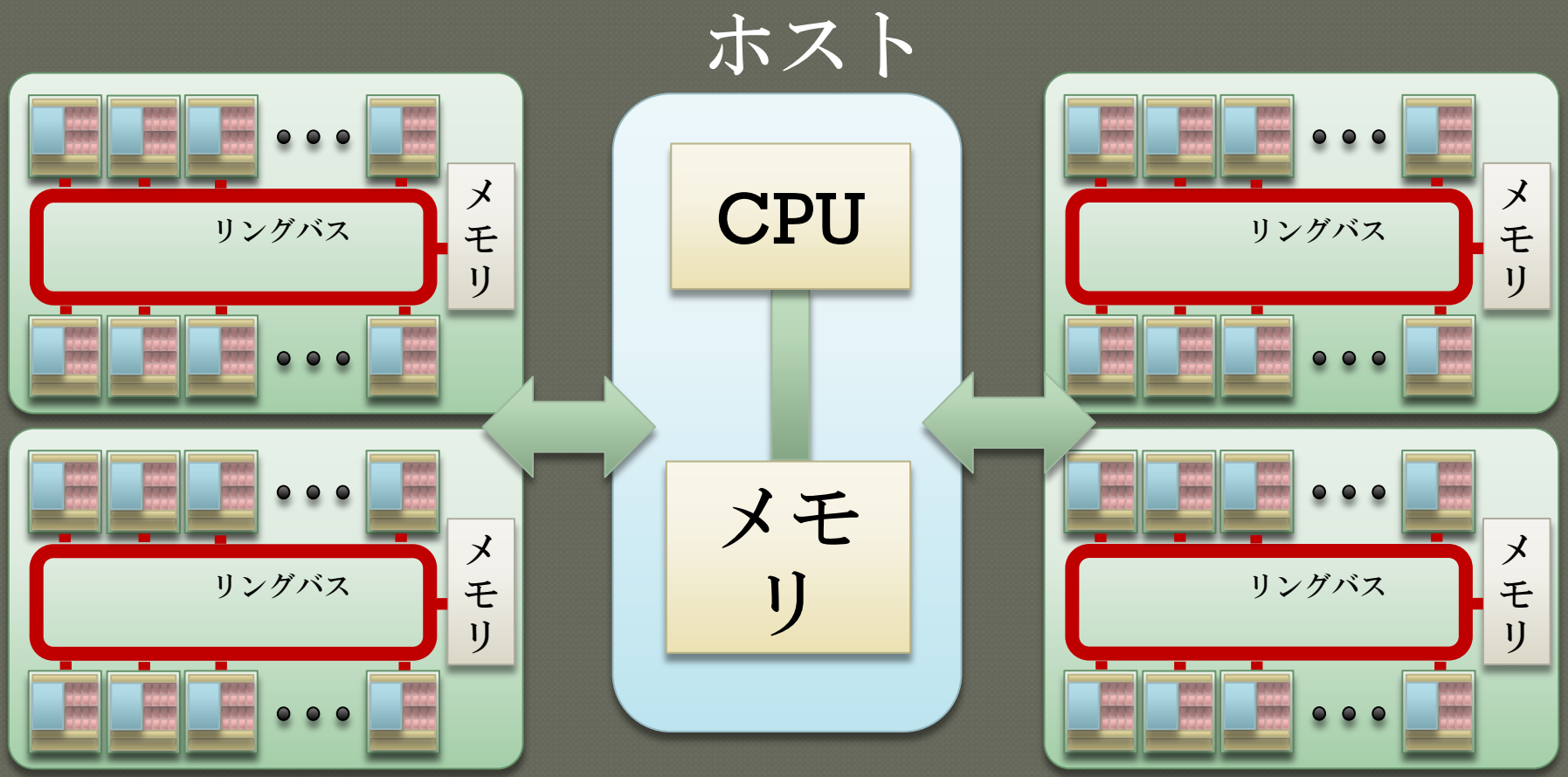
B　ホストとメニーコア“Larrabee”を接続



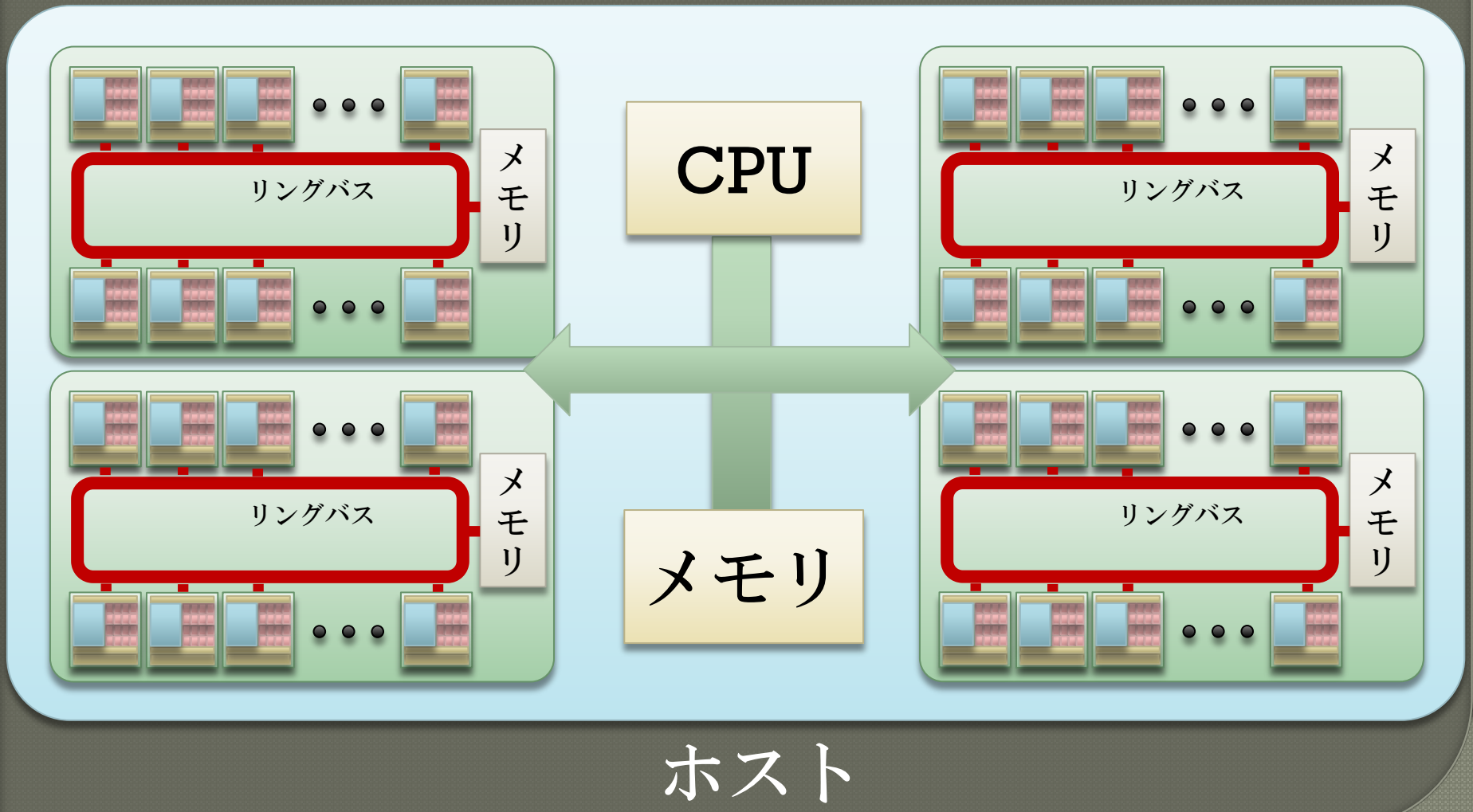
メニーコアチップ“Larrabee”中の１コア



B’ ホストとメニーコア“Larrabee”群を接続

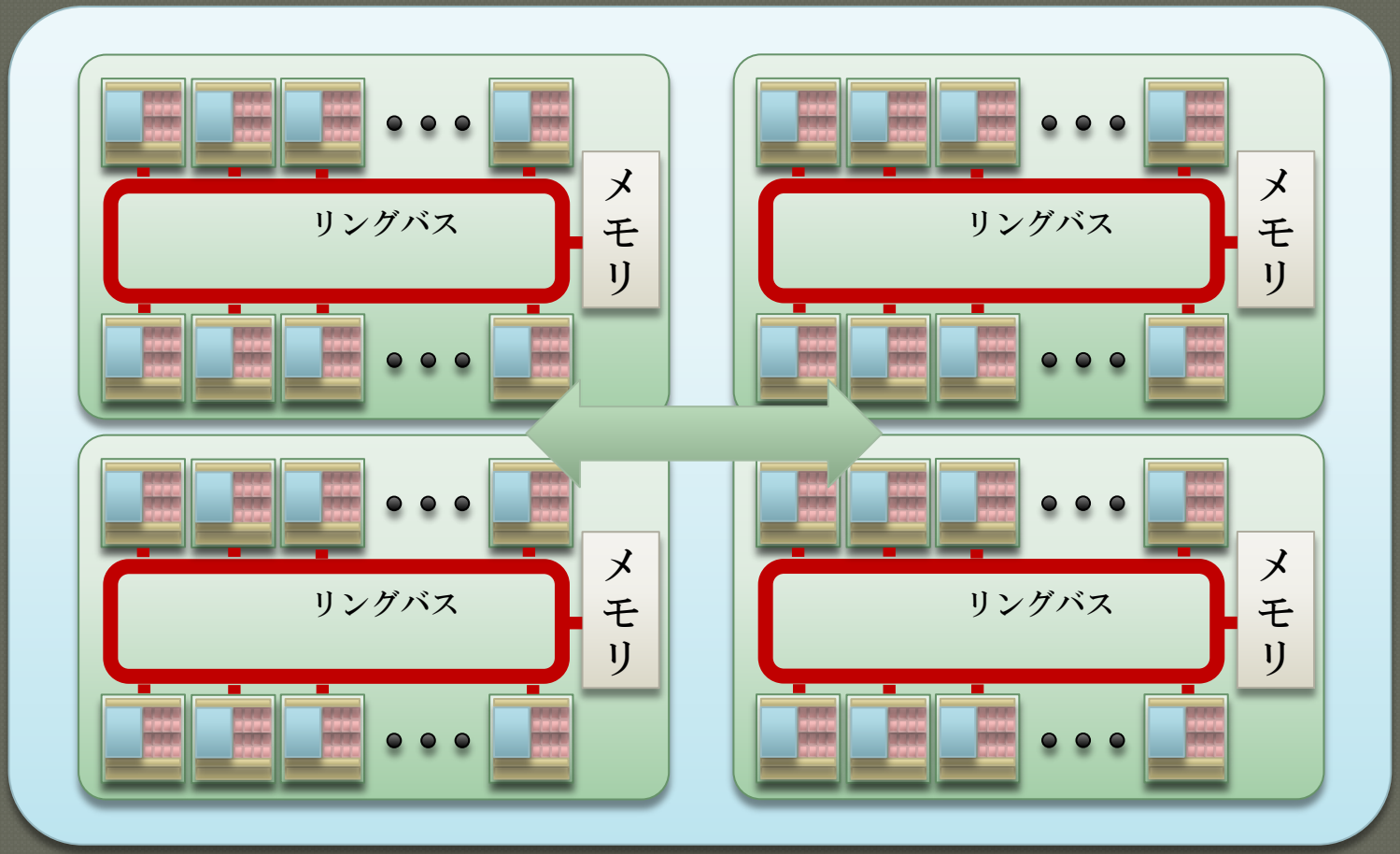


C メニーコアチップとホストを一体化



メニーコアのみでホスト(node)を構成

D

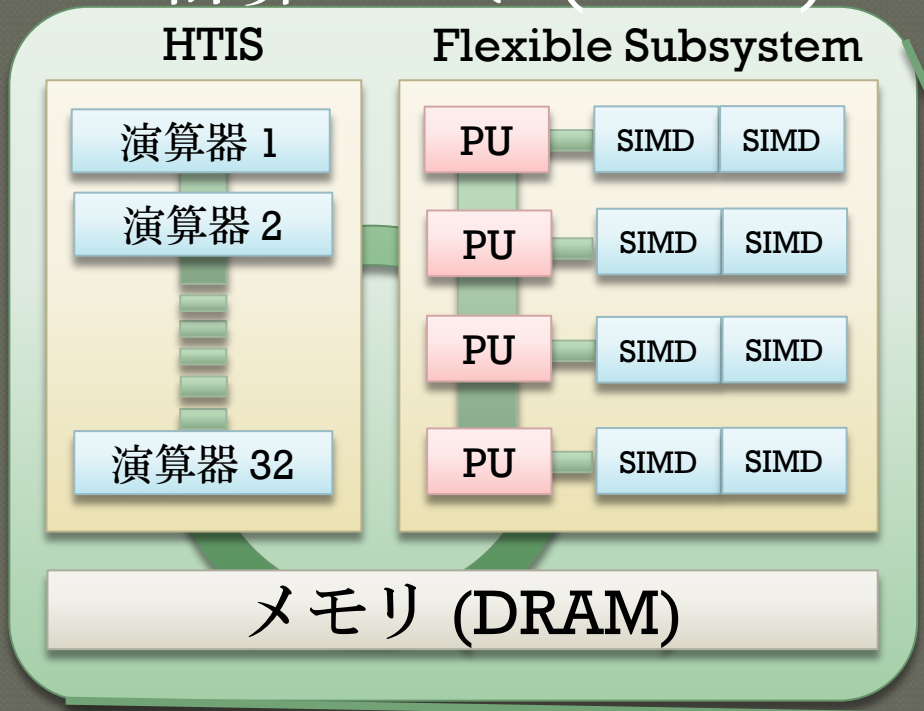


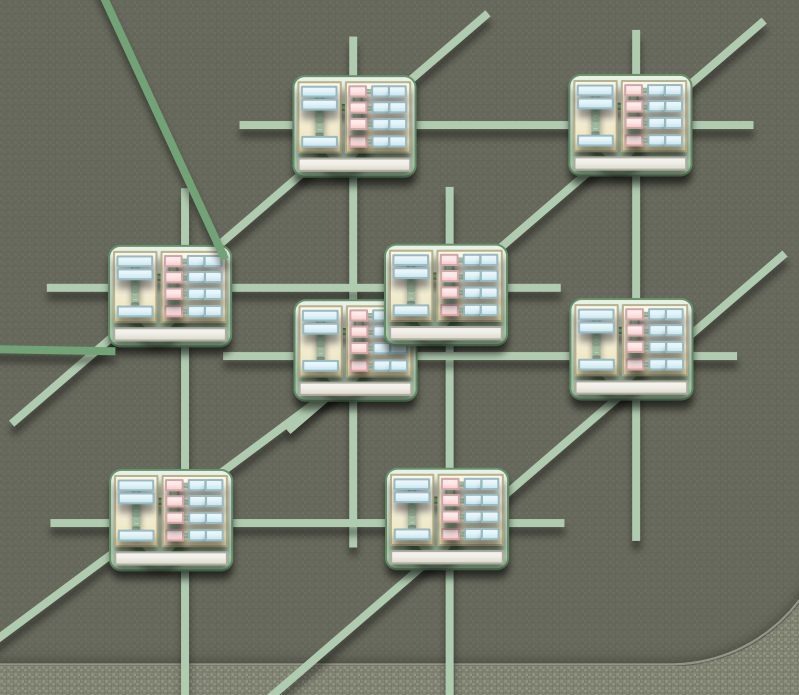
α 専用マシンの例: Anton

- MD (分子動力学) に特化したスーパーコンピュータ
  - 開発主体: D.E. Shaw Research
- 3D torus 状に結合された 512 の計算ノードからなる
  - 各計算ノードは MD に特化した 2 つのサブシステムとメモリ (DRAM) からなる
  - 2 つのサブシステムとも ASIC で作成されている
    - ASIC = Application-specific integrated circuit
    1. HTIS (High-Throughput Interaction Subsystem)
       - 静電気力・ファンデルワールス力の計算に用いる
         - 力の計算をパイプラインで行う 32 の演算ユニット (800 MHz) からなる
           - プログラミング不可
    2. Flexible subsystem
       - その他の計算 (FFT など) に用いる
         - 4 つの汎用計算コアと 8 つの MD の特化した SIMD 演算ユニットからなる (400 MHz)
           - プログラミング可能

α Anton の概略図

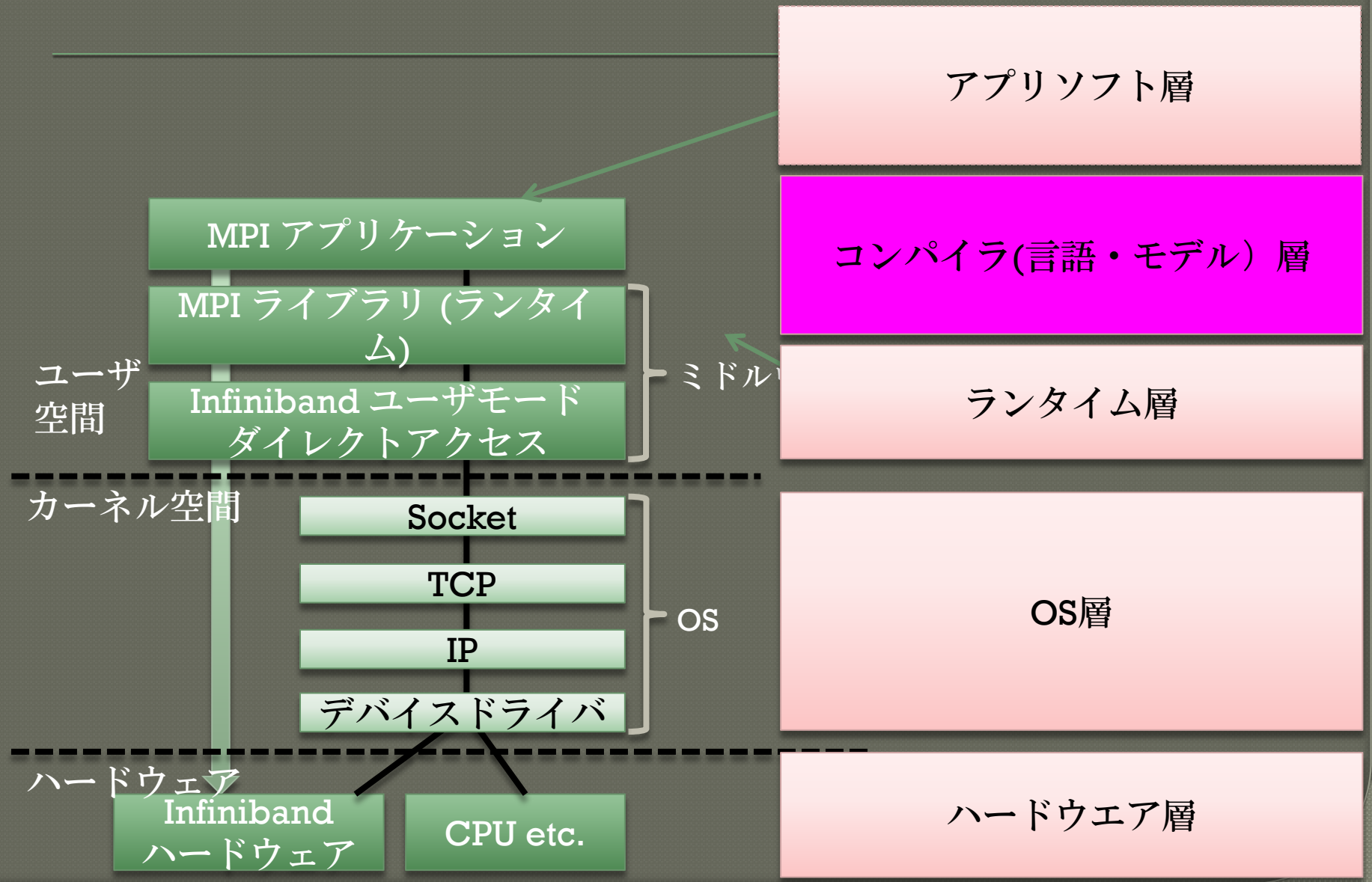
計算ノード (× 512)





ポストペタでシステムソフトはどうなるか？

ソフトウェアスタック


アプリソフト層
MPI アプリケーション
コンパイラ(言語・モデル）層
MPI ライブラリ (ランタイム)
ユーザ空間
Infiniband ユーザモード ダイレクトアクセス
ミドル
ランタイム層
カーネル空間
Socket
TCP
IP
デバイスドライバ
OS
OS層
ハードウェア
Infiniband ハードウェア
CPU etc.
ハードウエア層

Programming Models & Languages

- OpenMP + MPI
- XcalableMP(XMP)　- 主宰（佐藤三久）
- UPC, CAF(Co-Array Fortran)

- DARPA
  High Productivity Computing Systems (HPCS) Initiative

  アプリ作成・構築の生産性を重視

  プログラミングシステム開発を募集

  - X10 (IBM),
  - Fortress (Sun → Oracle),
  - Chapel (Cray)

- エクサスケール向けには、模索中

OpenMPとXcalableMP

- OpenMP
  - 共有メモリ環境での並列処理を記述するための
    C 言語, C++言語, および Fortran 言語拡張
  - Explicit parallelism
    - 並列処理は pragma(注釈) として記述されるため
      元の言語の構文に変更はない
      - シーケンシャルプログラムに注釈を追加する形

- XcalableMP XcalableMP specification Working Group（佐藤三久主宰）
  - 分散メモリ環境での並列処理を記述するための
    C 言語および Fortran 言語拡張
  - Explicit parallelism
    - 並列処理は pragma (注釈)として記述されるため
      元の言語の構文に変更はない
      - シーケンシャルプログラムに注釈を追加する形
    - 計算ノード間の通信は注釈を追加された部分のみで発生する
      - プログラマが実効性能を予測しやすい

UPC (Unified Parallel C)

- 言語の特徴
  - C 言語を並列実行機能で拡張した言語
  - Explicit parallelism
    - 基本的には SPMD モデル
      - 同一のプログラムを複数のスレッドで実行する
  - PGAS (Partitioned Global Address Space)
    - 1 つのプログラム中における全てのスレッドは 1 つのメモリアドレス空間を共有する
    - ただし、あるスレッドの「private」メモリは他のスレッドからアクセスできない
    - 一方、あるスレッドの「shared」メモリは他のスレッドからアクセスできるが性能は出ない
    - また「shared」メモリへのアクセスは race condition を生じる可能性があるので明示的にロックやバリアで同期する必要がある
- 想定実行環境
  - 並列・分散実行環境全般
- UC Berkeley, HP, IBM, Cray 等がそれぞれ実装している
  - 言語のデザインは UPC Consortium

X10

並列オブジェクトをベースとしているともいわれる

- 言語の特徴
  - Explicit parallelism
    - プログラマは明示的にスレッド (*activity*) の生成を指示する
  - PGAS (Partitioned Global Address Space)
    - 1 つのプログラム中において全てのスレッドは 1 つのメモリアドレス空間を共有する
    - ただし「*place* (場所)」の概念がある
      - 各スレッド (activity) が直接アクセス可能なデータは そのスレッドが属する place に保存されたものに限られる
    - 同一 place 内のスレッド間では race condition が生じる可能性がある
      - プログラマは atomic ブロックを用いるなどして明示的に回避する必要がある
  - スレッド間には親子関係がある
    - ロック同期の複雑性 (特にデッドロック) を回避するため
      - 親スレッドは子スレッドの処理終了を待てるが 子スレッドは親スレッドの処理終了を待てない
        - ただし「future」を使うとデッドロックが生じる可能性がある
- 想定実行環境
  - SMP マシンのクラスタシステム
- 開発主体
  - IBM (T. J. Watson Research Center)

Fortress

- 言語の特徴
  - Implicit parallelism
    - プログラマが明示的に並行実行処理を記述する必要はない
      - 実際に並行に実行されるかは処理系依存
    - (言語仕様としては) ほとんどの演算が
      並行に実行される可能性がある
  - 数学的記法にもとづくプログラミング
    - Unicode を積極的に利用することで
      プログラムを論文中の数式のように書くことができる→Exa向け？？？
  - Transactional memory
    - ロック同期の複雑性を回避するため
      Software transactional memory 技術を採用している
      - 複数のスレッド・プロセスが一貫性なく同時にメモリアクセスした場合、
        それらのスレッドの処理をやり直す
- 想定実行環境
  - 並列・分散実行環境全般
- 開発主体
  - Sun Microsystems → Oracle, 2007年よりオープンソース化

Fortress プログラムの例

```
component isqrt
import System.{args}
export Executable

⊛ Check that root is the integer square-root of x
inrange(x: ℤ, root: ℤ): Boolean =
    root^2 ≤ x < (root + 1)^2

⊛ isqrt using Newton's method; use the argument as the initial guess
isqrt(x: ℤ): ℤ = do
    approx: ℤ := x
    while ¬inrange (x, approx) do
        approx −= (approx − (x ÷ approx) + 1) ÷ 2
    end
    approx
end

⊛ Take the isqrt and print the result
run() = do
    getArg (i) = strToInt args_i      ⊛ local function definition
    println (isqrt (getArg 0))
end

end isqrt
```

プログラム中に並列化処理が
明示的に記述されていない

論文中の数式のように
プログラムを記述することができる

出典: http://projectfortress.sun.com/Projects/Community/wiki/IntegerSquareRoot

計算機科学に無理な要求をしないで

- 東工大・青木先生曰く、
- 私が学生の頃は、将来はコンパイラがすべて自動並列化をやってくれてプログラミングは楽になるだろうと思っていましたが、
- 現実はまったくの逆で、CPUでもマルチコアになっていますし、（GPUでも）性能を引き出すためには、プログラマがハードアーキテクチャをかなり意識してプログラムを書かないと十分性能を引き出せない状況になっています。
- ということで、昔、想像していた未来とは大きく違った環境になっているというのが今の心境です。

ポストペタ・エクサマシンは？

ポストペタ・エクサマシンの論点０

- 別にエクサマシンを目指さなくても？
  - 机の側に置ける１ペタマシンがあればＯＫよ！
  - つなぎたい人はつなげればいいじゃん！
  - 「クラウド」とかもっと発展するのでは？

- まずポストペタマシンから考えて作ってみましょう！？

ポストペタ・エクサマシンの論点

以下は間違いない！

- 演算性能　＞＞　メモリバンド幅　＞＞　ネットワークバンド幅
- Byte/Flops（データ転送能力と計算能力の比）の値が重要
- ノード間通信はあまり早くならないが、CPU-GPU一体化は進む
- processor間の同期不要の場合でも、 processor間の通信遅延は無視できない
  → 「遅延隠ぺい(latency hiding)]を越えた、新たなアルゴリズムが必要

一方、

- 数値計算中心かのマシンか？/大規模データ処理機能はどうするか？
- 汎用マシン中心か専用マシン中心か？ heterogeneousかhomogeneousか？
- 日本製チップから起こしてゆくか、commodityチップを多用するか？
- 国際分業体制を取るか・取れるか？

ポストペタに向けた日本の研究

- 計算科学研究機構での研究
  - ミドルウエアチーム（石川）
  - 言語・開発環境チーム（佐藤）
    - アプリの検証(モデル検証)も含む
  - （専用）プロセッサチーム(泰地)

  予定
  - （汎用）プロセッサシステムチーム
  - 数値計算・アルゴリズム研究開発チーム
  - 超大規模データ処理研究開発チーム

- エクサに向けたその他の取り組み
  - JST CREST
    「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」

ポストペタ・エクサへの放言１

- 中国のtop500奪取で、
  - 米国にスプートニク衝撃
  - アポロ計画に類似するの体制・エンジニアリングが必要
- 専用機なら100ペタは難しくない
- エクサは、ソフトウエアが問題、
  - BlueGenesの性能予測では 100p, 1エクサは簡単言ってる
  - メモリバンド幅やinterconnectがスケールしない

    → このような大きな問題のつけがソフトへ
- エクサには、アルゴリズムの変換・革命が必要?
  - 例： FFTはスケールしない（メモリのグローバルアクセスが必要）

ポストペタ・エクサへの放言２

- 日本でデバイス技術投資に匹敵する投資がプロセッサ開発にされてもいいはず！
  - 例えば、Mainstreamである「インテルcore i 7」はすごい、負ける
- インテルやＩＢＭはエクサにコミットするか？
  - だから、日本独自ものをやるべし
  - ＨＰＣに収益性がないというのは誤り
  - 日本のメーカはmainstream (highとlowの中間) で覇権はとれない、むしろ high end のマーケットなら勝てる。
- 目的や大義がないとエクサにチャレンジ不能

ポストペタへの国内の体制は？

- 早急に体制を整え始めないと間に合わない
  - シンポを企画、審議会を立ち上げ？
    - 国内のサポート、雰囲気の醸成
  - 文部科学省でも、来年度はじめには次々世代に向けたシンポを開催予定
  - 「京」で早急に成果を上げていかないと
- 段階的にポストペタスケールマシンを構成・計画していくべき
  - オープンの議論でアーキテクチャを決めて行く
  - しっかりとしたロードマップを作る。
  - ハードウエアも 2, 3 ステップを踏んで段階的に強力にして行く
  - システムソフトも大学とメーカ (3 社？) が同じチームに入って構築
- アプリとのco-designが必要
  - DOEはExascale Co-design Centerを作り始めた！

  ハード、ソフト、数値計算、アルゴリズム、アプリが co-design する
- エクサスケールは１つのアイデア（1つのチップ）では出来ない
  - アプリの要求を十分抽出し、それを分解して検討してゆく
  - （一方、アプリ側の言うことは聞くな、という考えもある）

終わり