

金融工学とGrid Computing

野村證券株式会社

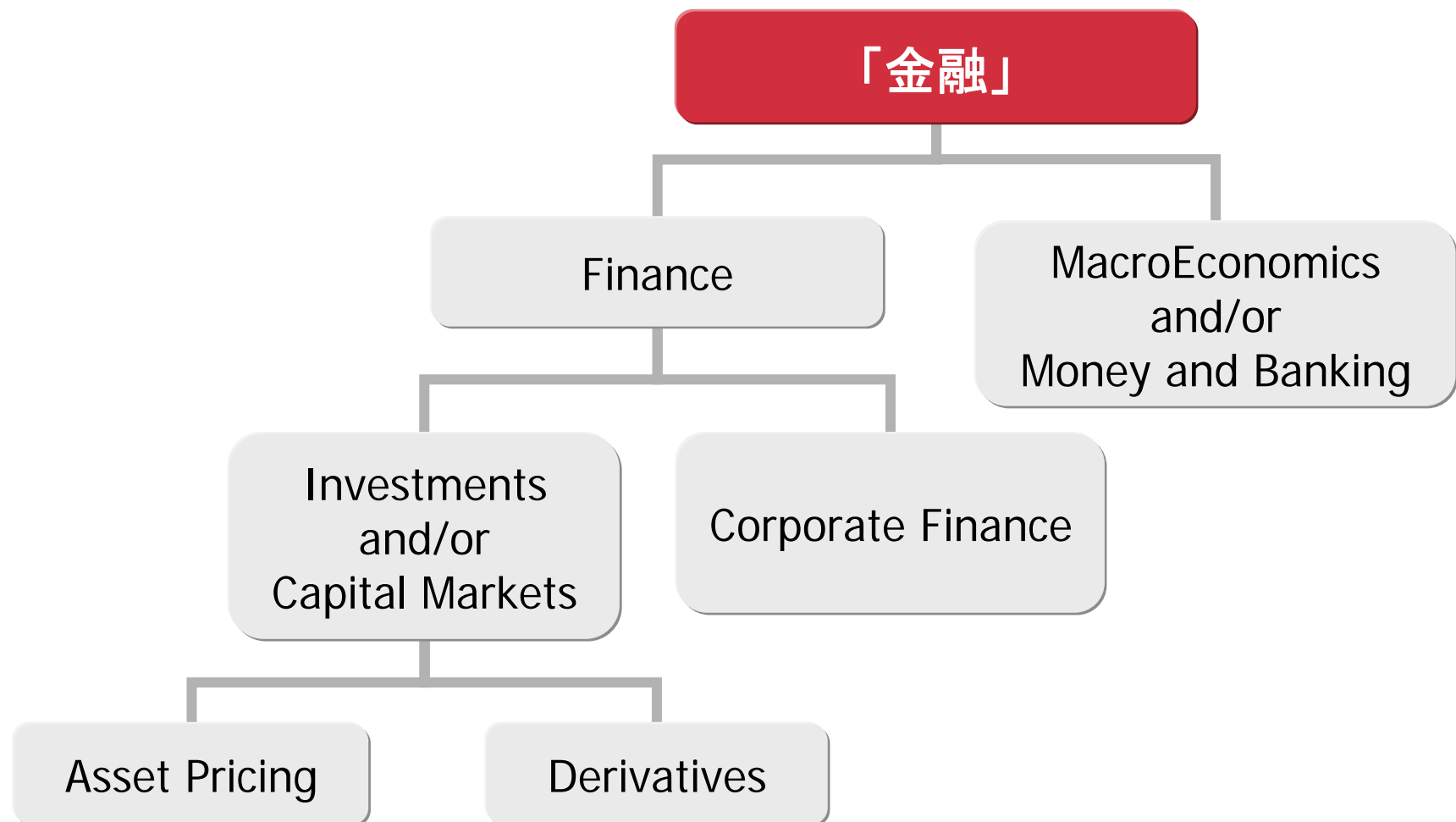
金融工学研究センター 債券投資戦略グループ

クオンツアナリスト 野村 俊朗

0. 金融工学とはそもそも？



「金融」関係の学問体系



米国のFinancial Engineering

「過去20年の間に、金融市場に関する理論・実証分析は飛躍的に進歩した。このような学問上の進歩は、次第に現実経済における、多様な新しい金融商品・金融サービスの爆発的な増加をもたらし、それに伴うリスクとリターンを管理するための、洗練された数量的な分析ツールの必要性を生み出した。そのようなツールは、今日のプロの投資家やファイナンシャル・マネージャーにとって必須のものとなり、その結果“Financial Engineering”と呼ばれる、明確な知識の体系(well-defined body of knowledge)を形成するに至った」

by Andrew Lo (1998), “Overview of The Track in Financial Engineering”, 祝迫訳

つまり、

- 「実務家の側 (Practitioners) の立場から見て必要な知識、特に数量的なスキル」
- 「数量的な側面を重視した目的志向型のファイナンスという学問の実践」

学問としての研究のベースは

- Mathematical Finance
- Computational Finance など

日本の「金融工学」

「資産運用や取引、リスクヘッジ、リスクマネジメント、投資に関する意思決定などに関わる工学的研究全般」

— by Wikipedia

つまり

- 米国でいうFinance (+ Macroeconomics?) の工学的アプローチ

学問としての研究のベースは

- 金融経済学 (Financial Economics)
- 数理ファイナンス理論 など

応用先として

- 投資銀行における企業価値の測定
- デリバティブ(先物、先渡、オプション)取引
- 機関投資家の最適投資戦略
- 不動産担保証券などのプライシング
- リアルオプション (Real Options Analysis) によるプロジェクト価値の測定
- 金融機関のリスクマネジメント など



1. 最近の金融業界の方向性



データ量・計算量が増えている

Intra-day / Tick データの利用

- 分析に必要なデータが膨大になる
- 新しい情報を反映して分析が必要となり、リアルタイム処理を求められる
- こういったニーズに耐えうる計算 / 分析システムの開発へと発展

種々のケースを想定したリスク シミュレーションの要請

- ほとんど同じだが少しだけ与える情報や条件が異なるシミュレーションを多数行う
 - 世界的に見ればこういったリスク シミュレーションは Mark to Market の方向に向かっている
- 計算対象(量)の増大
- 計算回数増加
 - 半年に1回 → 四半期に1回 → 月1回 → 週1回 → 毎日 → 1日2回 → 1時間1回 → ...

複雑な組み合わせの商品の開発

- 競争の激化で、単純な商品から種々の商品を組み合わせたものに発展
- 商品特性や最適解探索などがますます複雑化、計算量の増大

スピードアップが必要となってきた

これまで: 割とゆっくりだった金融の現場のスピード

例: 手書き帳票や場立ち

- 「手でやった方が話が早い」

最近: 人間の感覚を超えたスピードに

例: 電子取引、Straight Through Processing、T+0 決済、24時間対応

- 人間の手や判断では追いつかない

なるべく早い反応が求められる

- アルゴリズムトレード
- リアルタイムOS
- ファイルの出し入れまでもが「余計な時間コスト」

- この解の 1 つが Data Grid



2. 最近のプログラムのテクニック



Multi-Core CPUのためのTLP

ILP から TLP への移行

■ Instruction-Level Parallelism (ILP)

- CPUが命令レベルで並列実行をしてくれる
- 命令レベルでの並列には限界がある: 命令実行制御が大変
- 設計コストの増大、性能向上の頭打ち

■ Thread-Level Parallelism (TLP)

- Trace Level (数命令) → Thread Level (複数プロセス)
- より粒度の高い、最低実行単位としての Thread の並列化
- 背景には、共有メモリ、共有レジスタの概念



CPU も Single-Core から Multi-Core へ

- 半導体集積、動作スピード向上に伴う熱問題
- 製造プロセス微細化に伴うリーク電流の増加
- より低い動作スピード、電圧などのコントロールをしても処理性能が稼げる
- ただし、CPUの面積が大きくなり、歩留まりは低下する
- 単一コアで見た場合の性能は低下する

分散処理を前提としたロジック

並列化手法	長所	短所
データ並列化	<ul style="list-style-type: none"> プログラムを作成しやすい 読みやすいコードがかけられる 効率が良い <ul style="list-style-type: none"> 分割データ間での依存性がない 処理時間に偏りがない場合、処理中の同期コストがないため 	<ul style="list-style-type: none"> 分割データの処理時間に偏りがあると並列度が下がる 性能を上げるには、各部分のデータの処理時間を適切に見積もる必要がある
分割統治法	<ul style="list-style-type: none"> 部分解に分解して解く場合に適用 自然な形で実装、並列化しやすい 	<ul style="list-style-type: none"> 再帰呼び出しによって実装のため <ul style="list-style-type: none"> デバッグしにくい 処理を追っていく
タスク並列化	<ul style="list-style-type: none"> 各タスクの処理時間の偏りを許容して適当な並列度を得ることができる <ul style="list-style-type: none"> 個々のタスクの粒度が適度に小さい場合 ソースコードが比較的わかりやすい 	<ul style="list-style-type: none"> タスク マスタ、タスク キューに処理が集中 → ボトルネックになる可能性 スケジューラ次第では性能が得られない <ul style="list-style-type: none"> 複数のスレッドがキューを介して協調 スレッド・キューの調停が必要
パイプライン並列化	<ul style="list-style-type: none"> ストリーム処理で解く場合に適用 自然な形で実装、並列しやすい 	<ul style="list-style-type: none"> 1データに対する処理時間は長期化 <ul style="list-style-type: none"> 処理ステップごとにキュー操作が入る 高い並列度は得にくい <ul style="list-style-type: none"> 同期によるスレッド切り替えが複雑に発生 タスク並列化と同様の短所を持つ

データ並列化

概略

- N人の作業員/1つの大きなデータの固まり
- データの固まりをN等分し、作業員が同時に処理

利点

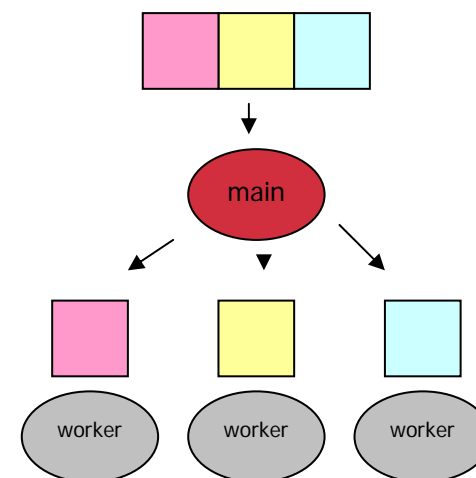
- 単純
- プログラムの見通しがよい
- 得たい並列度に合わせてデータ分割するだけ
- それぞれのスレッドの処理完了を待つだけ
- 同期・協調のオーバーヘッドがほとんどない

欠点

- 各スレッドが処理するデータ範囲を設定しなければならない→計算量を平均的に分散させるのが難しい
→想定した並列度や性能が得にくい
- 各スレッドが独立に作業をするため、相互依存関係はNG

ポイント

- 処理時間の偏りがないようなデータ分割の工夫



分割統治法

概略

- 問題を木構造に分割して部分解を求め、統合して解を求める場合に適用
- 分割フェーズ、統治フェーズを繰り返し実行
 - 分割フェーズ: 処理データを徐々に細分化、適当な並列度が得られるまで
→その後はデータ並列化と同じ
 - 統治フェーズ: 分割フェーズと逆の順番でデータ部分を結合

ポイント

- 基本的にデータ並列化と同じ
- データ並列化と利点・欠点も同様
- データ並列化を再帰呼び出ししている為、デバッグ等は大変
- 逐次プログラムでも実現されている手法である
 - 部分問題を並列に解かせるか、逐次求めるかの違い

タスク並列化

概略

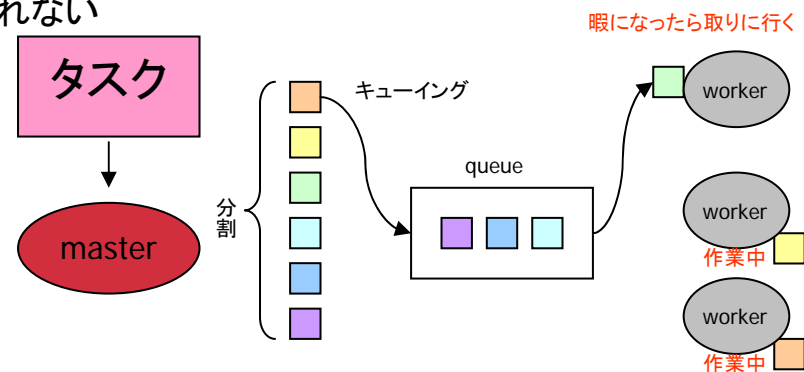
- 処理を「仕事」単位のもとまりに分けて分割
- マスタ・ワーカ・キュー(マスタが管理して使わないケースもある)によって実装

利点

- 各スレッドの処理時間のばらつきを抑える機構が実現しやすい
- 結果的に適当な並列度をもつプログラムを作成しやすい

欠点

- キューイング(enqueue/dequeue)の待ち時間コスト/オーバーヘッドが発生
- 待ちスレッドに対するスケジューリング方法による効率低下
- タスクの粒度次第では処理時間のばらつきが抑えられない



パイプライン並列化

概略

- 「流れ作業」による方法
 - データ要素に「一連の加工」をおこなう
 - 加工を担当する部分(ステージ)を分割している
- 1要素処理時間は変わらない
- 次々とデータ処理する場合に並列化効果が得られる

利点

- ストリーム処理できるアルゴリズムを直接プログラム化できる
 - 入力データに複数処理を適用するマルチメディア系のストリーミング処理など
- スループットは逐次処理よりも向上

欠点

- 1データについての処理についてはオーバヘッド(キュー操作、ステージング管理等の同期・協調スケジューリング)の分、逐次処理よりも時間がかかる
- パイプラインの各ステージの処理時間に偏りがあると並列化効率が落ちる

どう並列化するか

プロファイリング

- 各部分の性能(処理時間)計測・挙動やボトルネックの把握
- どこを並列化するべきかを探索

並列化の戦略と性能予測

- 最適な並列化方法の検討
- 必要であればデータ構造・アルゴリズム自体の変更
- 並列化後の性能予測(理論上・コスト勘案なし)

実装

- 実際のプログラム設計と実装

計測と解析

- 実装結果の処理時間計測&予測との比較

並列化の限界

アムダールの法則

- 速度向上度(n) = $1 / ((p/n) + (1-p))$
 - n: プロセッサコア数
 - p: 並列化を行った処理の処理時間 / 全体の処理時間



並列化のコスト(1-pが0に近づかない)

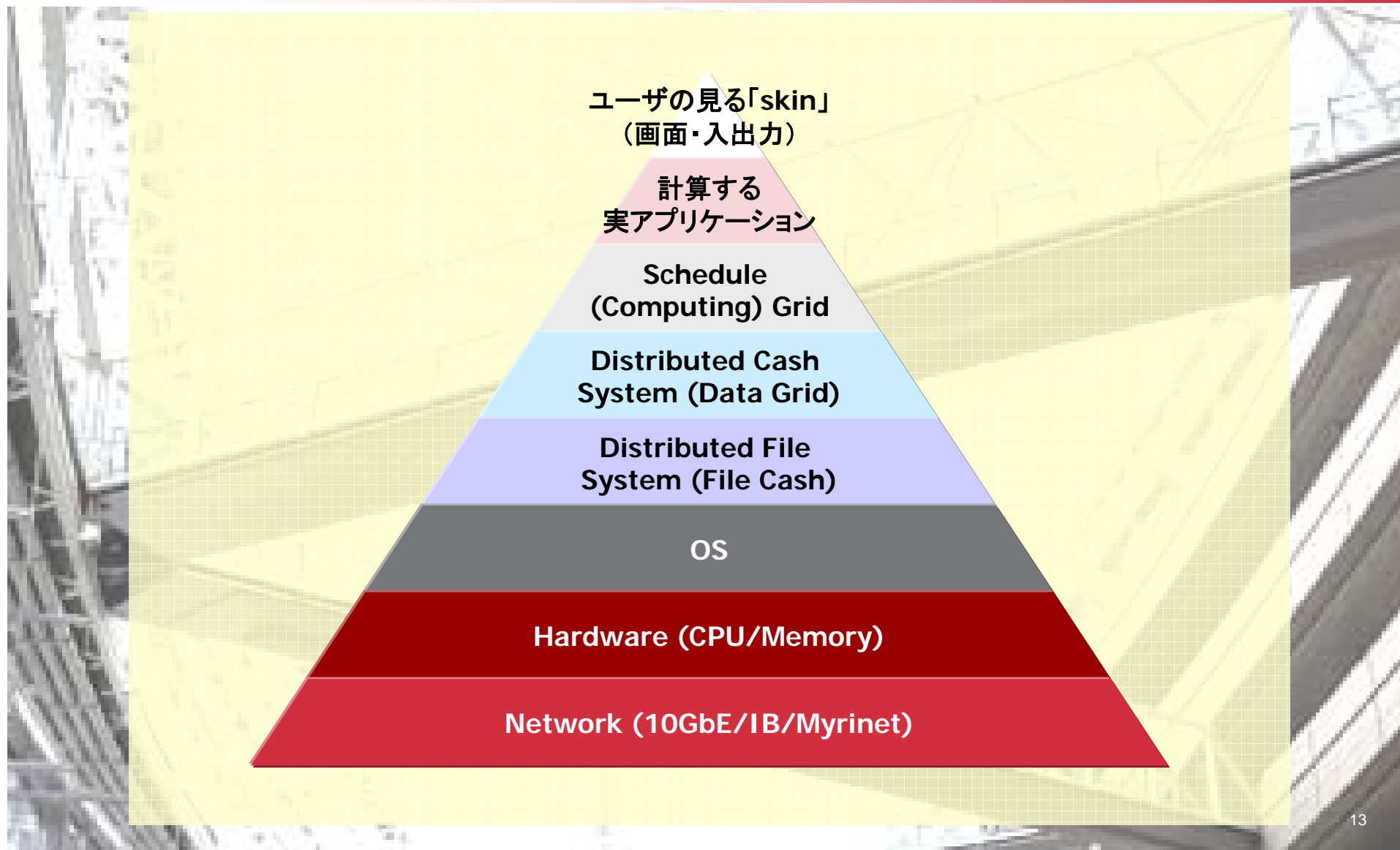
- 同期処理 / 排他制御が必要
- スレッドなどの負荷の偏りがある
- スレッドの生成コストがかかる
- 並列化の前処理・後処理のコストがかかる

アムダール則の趣旨: 並列化プログラムの速度向上度は、対象となるプログラム中の並列化できない部分の割合に大きく左右される

3. 金融工学とGrid



Gridの階層モデル



金融工学の方向性とGrid

データ量や計算量が増えている:アムダール則に従った対応

- n をいかに大きくするか:
 - 複数CPUを利用した計算システム
→ 複数のマシンにまたがる大規模分散計算システム
 - 大きくした n を下げない努力も必要
→ Schedule Grid など、スケーラブルかつ管理が容易な形で計算コアが追加できる環境への発展
- p をいかに 1 に近づけるか (並列化で解決できる部分)
 - シングルスレッドからマルチスレッド対応のプログラムへ転換
 - 並列度を高めるテクニックの実装
- $1-p$ をいかに 0 に近づけるか (並列化で解決できない部分)
 - アプリケーションinput/output コスト削減: Data Grid や In memory DB の利用
 - ネットワーク・トランザクションコスト削減: 分散ファイルシステムやInfiniBand/10G Ethernetの利用

計算量が少なくてもスピードアップが必要となってきた:

- $1-p$ をいかに 0 に近づける (並列化で解決できない部分) 努力が必要
- 基本的なロジックから根本的に改良する
 - 「ある程度の精度」を保ったままスピードが速い計算手法の開発:
乱数発生機構の改良 (含む、準乱数の利用)、収束しやすい計算アルゴリズム、計算量を少なくするための数学的アプローチ etc.

インフラ管理方法の発展と仮想化

ユーザは基本的に贅沢

- 早くて(計算や I/O)
- でかくて(メモリや HDD)
- 安くて(予算が！)
- 落ちない(計算が終わらない！お客さんが怒ってる！！)

本当に必要なのは落ちたことがユーザに認識されないシステム

- 落ちたらあきらめる
 - 「しょせんは分析システム、本番じゃない」「マシンが落ちたと言い訳しよう」
- 落ちても、早く上がってくればOK
 - 「予備マシンがあるからそっちでやろう」
- 複数台落ちても、どこかで動作している
 - 柔軟な仮想化：Gridもこれと同等の環境を提供
- まったく落ちないシステム
 - 絶望的：費用、監視、技術...



金融工学とGrid / 並列化の関わり

金融科学ではなく金融工学

- 高度なモデルをつくれれば終わりではなく、実体化が必要
- 現実：スピードアップ、データ増大、モデル複雑化、技術のコモディティ化
- キャッチアップするための High Performance Computing (HPC)

かつてロケットを作っていたエンジニアがウォール街に流れてきたのが金融工学のはじまりだったー

- 金融工学でも、その源流である航空宇宙工学でHPCが必要になったようにHPC化が不可欠になった
- ロケットエンジニアがそうであったように専門化も進行中：
理論、モデリング、デザイン、プログラミング、システムインフラ etc.

グリッド(仮想)化、並列化によるパラダイムシフト

- 金融の現場で動いているプログラムは、(意外と?)泥臭いつくり
- システムを構築時の考え方を強制的変更
- 並列化向けの設計にアーキテクチャを見直さざるを得ない

4. 結論



結論

金融工学とは結局何なのか？

- 日米で捕らえ方が違う：使いこなし方の違い
- 金融業＋金融工学 → 金融製造業 → 金融装置産業？

システムは使うもの

- 欲しいのは『早くて(計算/IO)、でかくて(メモリ/HDD)、安くて、落ちない』理想のマシン！
- どうやってこれを実現するか？

考え方のパラダイム シフト

- 直列的なロジック → 並列的に
- 秒単位のスピード認識 → マイクロ秒単位
- リニアなメモリマップ / 行指向、CSV 的なデータ
→ 多次元で広大なデータ空間 / 多対多の関係を持ったデータ

Data Grid をベースとした仮想化が進めば、ワンマシンのメモリマップの枠から開放され、思考の壁が取り払われる