ソフトウェア最適化方法論
パフォーマンス・サイクル

コースの目的

1. パフォーマンス・チューニングにおける重要な作業を識別する
2. パフォーマンス・チューニング用語を理解する
3. インテル® ソフトウェア・ツールが活用できる状況を把握する

コースの内容

- パフォーマンス・サイクルの概要
- パフォーマンス・サイクルの詳細
- まとめ

コースの内容

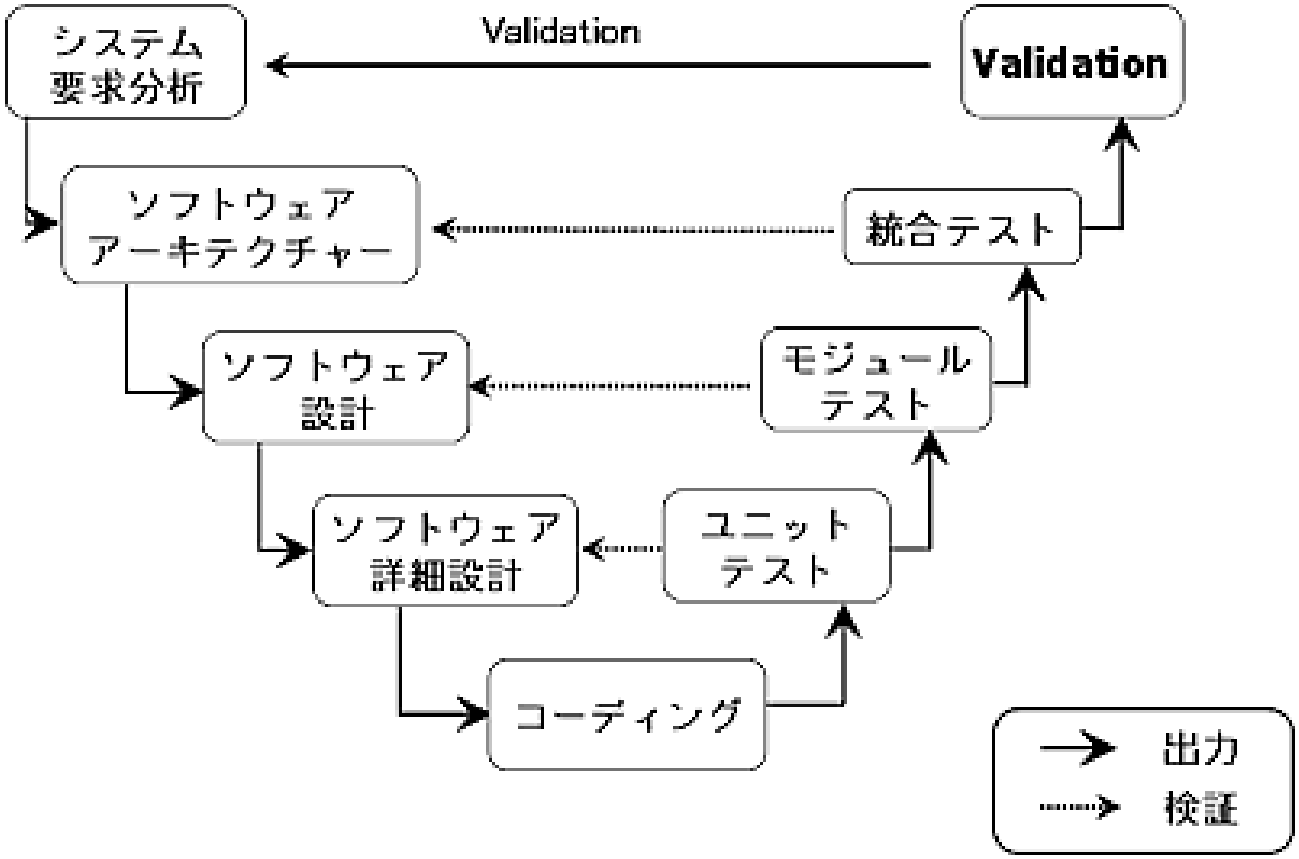
パフォーマンス・サイクルの概要

- パフォーマンス・サイクル
- 開始時期
- パフォーマンスの向上
- 終了時期
- 広い視野での再考察

パフォーマンス・サイクルの詳細

まとめ

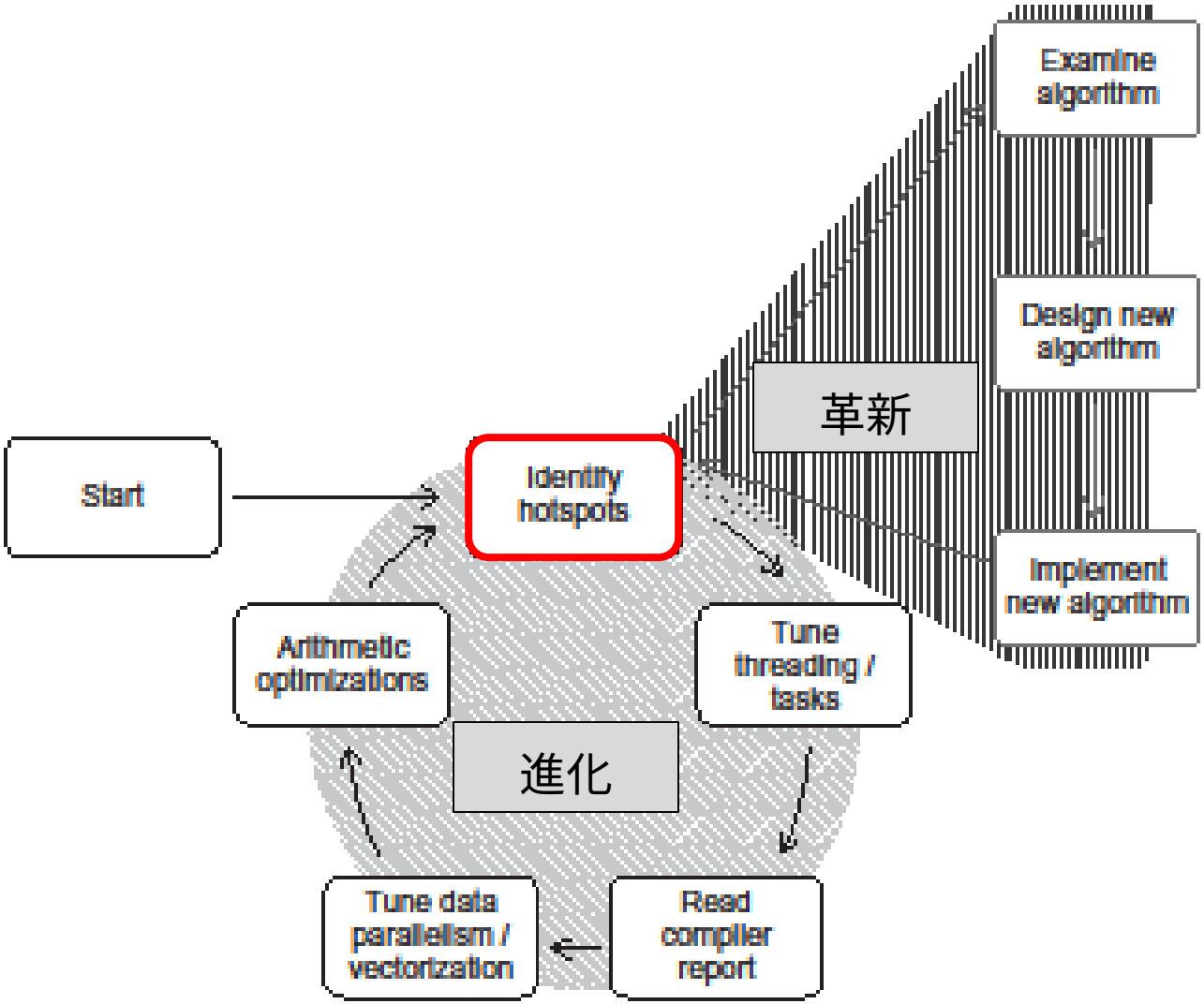
設計段階にパフォーマンス要件を取り込む



出典: IDAJ CAE 用語集

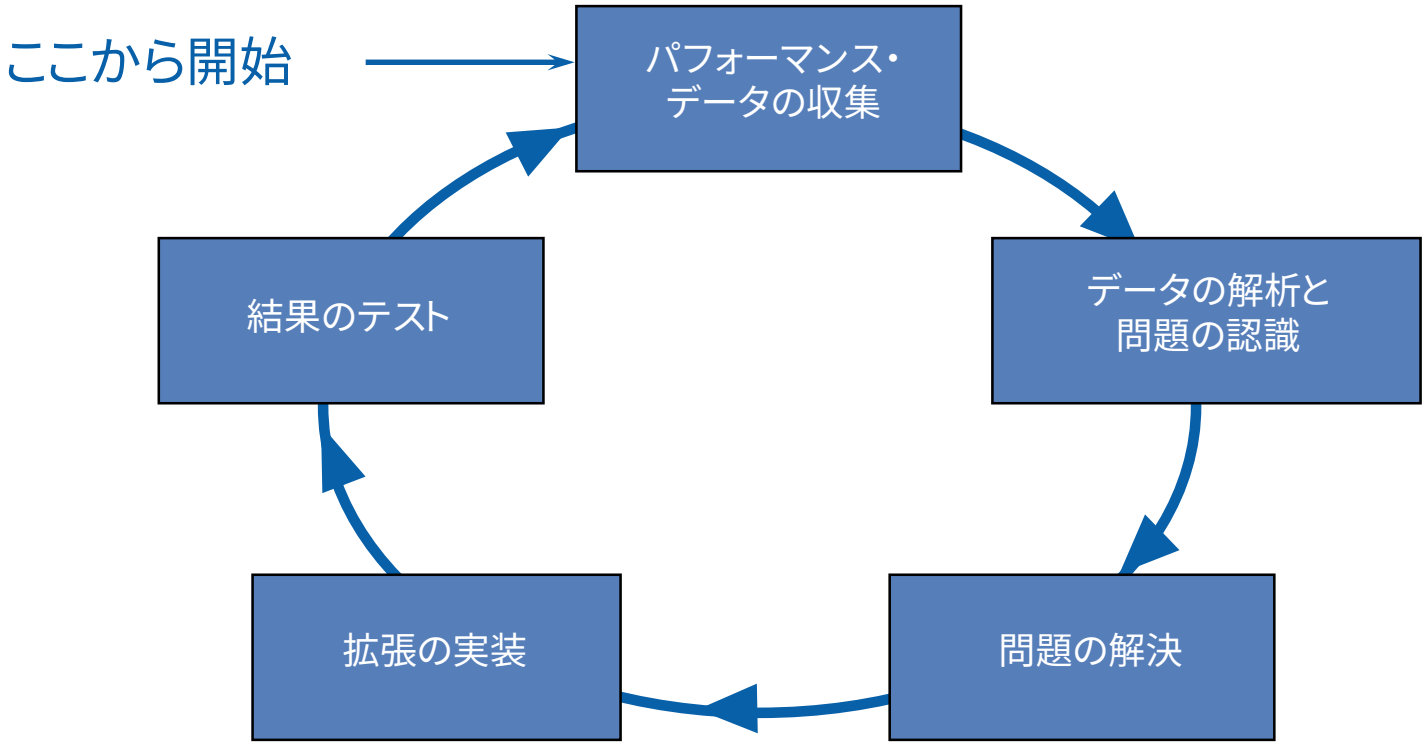
概要

最適化サイクル


Examine algorithm
Design new algorithm
Implement new algorithm
革新
Start
Identify hotspots
Arithmetic optimizations
Tune threading / tasks
進化
Tune data parallelism / vectorization
Read compiler report

概要

パフォーマンス・サイクル



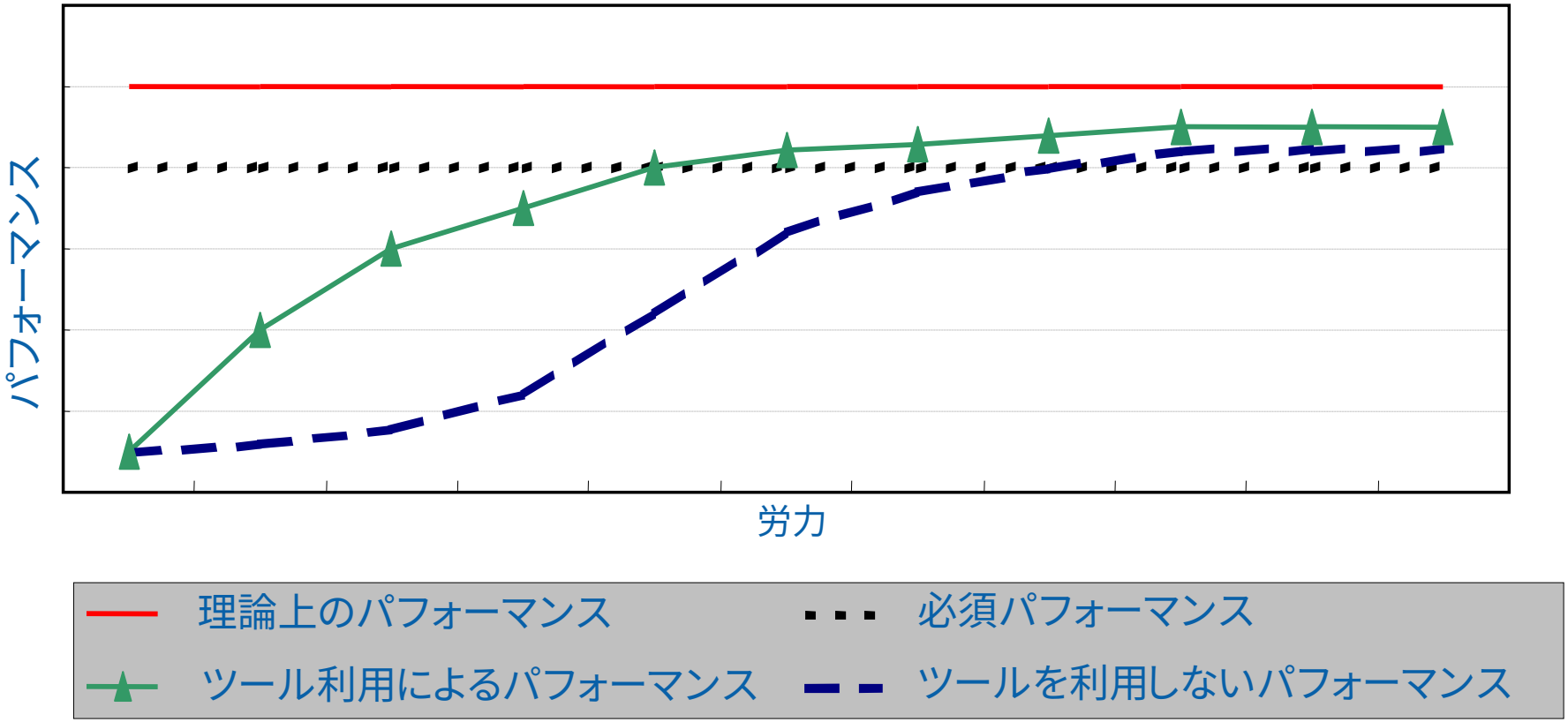
概要

開始時期

- ユーザーの要求
- ソフトウェア・ベンダーの要求
- 関連するドキュメントへパフォーマンス要求を盛り込む
- 製品ライフサイクルの各段階（要求事項確認、設計、およびテスト）でパフォーマンスを考慮
- 例外: アプリケーションのデバッグが完了し、ソースコードの整理が完了するまで “コード・チューニング” は行わない

概要

労力 vs. パフォーマンス



概要

終了時期

アーキテクチャーの利用効率は最大か?

理論的な最大値の計算方法を知っておく必要がある

パフォーマンス要件は満たされているか?

完了まで多岐にわたる最適化が徐々に行われる

チューニングのゴールを決定することは、すべてのチューニングの出発点です。すべてのチューニング戦略は、ターゲットとなるプロセッサー上で最短時間で最良のスピードアップを達成するというゴールに進むためであり、非常に重要なことです

概要

広い視野での再考察

「個々の細かな効率のことは忘れて、全体の効率について考えるべきです。
中途半端な最適化はよい結果を生みません」

Donald Knuth

質の高いコードとは:

- 可搬性がある
- 判読可能である
- メンテナンスが容易である
- 信頼性が高い

パフォーマンスと品質の最適なバランス

コースの内容

パフォーマンス・サイクルの概要

パフォーマンス・サイクルの詳細

- パフォーマンス・データの収集
- データの解析と問題の認識
- 問題の解決
- 拡張の実装

まとめ

パフォーマンス・サイクルの詳細

パフォーマンス・データの収集

タイマー

- 実際の利用時間を取得
- 正確で、オーバーヘッドが少ない

ツールを利用（インテル® VTune™ Amplifier XE等）

- プロファイラー: コードの使用率に関する情報を収集
- パフォーマンス・モニター: システムリソースの使用率に関する情報を収集

パフォーマンス・サイクルの詳細

適切なワークロード

良いワークロードには、次のような特徴がある

- 測定可能
  （性能計測が定量化できること）
- 再現可能
  （異なる実行で同じ結果をもたらす）
- 静的
  （仕事量が時間によって変化しない）
- 典型的
  （実行される作業は正常なシステム状況下におけるストレスを示す）

パフォーマンス・サイクルの詳細

データの解析と問題の認識

1. 現在のパフォーマンスの基準
2. hotspot の調査　（処理時間が費やされる場所）
3. ボトルネックの識別　（性能低下の原因）
4. 潜在的な最大パフォーマンスの計算

パフォーマンス・サイクルの詳細

hotspot の調査

パレートの法則（80/20 の法則）

- 重要でない 80% ではなく、重要な 20% に集中する

hotspot: アクティビティーが大量に含まれている
アプリケーションまたはシステム内の場所

通常はループからなる

hotspot のないアプリケーションでは、次の項目を検査する

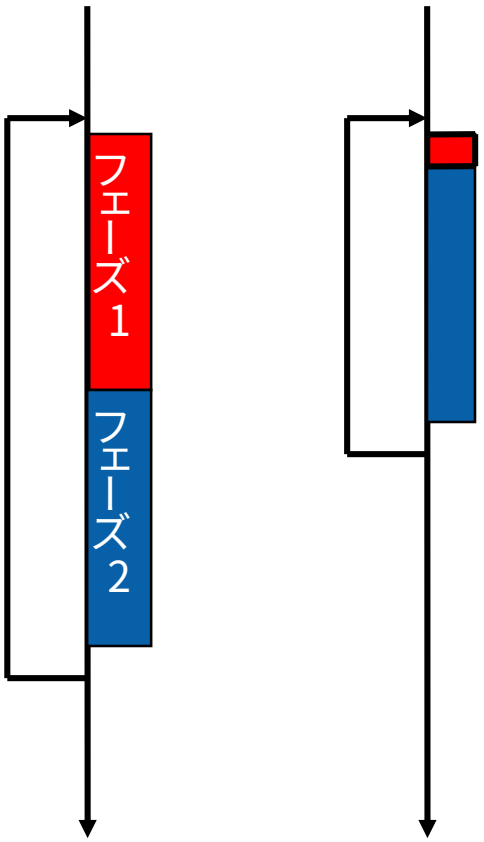
- メモリーレイアウト
- 例外
- コンパイラーの効率

パフォーマンス・サイクルの詳細

アムダールの法則:
図説

実行時間の半分のみが最適化された場合、2 倍のスピードアップは不可能である

フェーズ 1　＝　10 行のソースコード
フェーズ 2　＝　500 行のソースコード



パフォーマンス・サイクルの詳細

その他の考慮事項

- Big O 記法　（O（n）、O(n^2)、O（n log n））
- 使用率、効率、スループット、レイテンシー
- ボトルネック
  - I/O、メモリー、CPU
- MIPS/FLOPS/CPI
- 同時性、並列性
- スケーラビリティー
- 計算ごとのロード/ストア

演習 1:
フェルミの計算

```
float *loop(float *data1, float *data2) {
      float Pi = 3.14, *ans;
      ans = (float *) malloc(MAX * sizeof(float));
      for ( i = 1, i < MAX, i++) {
            ans[i] = data1[i] + Pi * data2[i]; }
      return ans;
}
```

計算:Big O 記法 縦軸に時間、横軸をnとした場合、データがすべてメモリ上で処理されるのであれば、リニアに右肩上がりとなる。

合計メモリー使用率 MAX＊float（4バイト）＊3つのデータ配列。

ストア vs. ロード vs. 浮動小数点演算 x ストア+x ロード+x 演算

演習 2:
フェルミの計算

MAX=4,000,000 (4 百万)として想定する

では、第 3 世代インテル® Core™ i7 3370 プロセッサー(3.40GHz)でシングルスレッドが計算を完了するための予想時間は？

ヒント：プロセッサーの動作クロックは最重要か？

4,000,000 * 4 byte * 4 load / store = 64,000,000 (64MB)
0.064 GB / 25.6 GB = 0.0025 sec

コースの内容

パフォーマンス・サイクルの概要

パフォーマンス・サイクルの詳細

- パフォーマンス・データの収集
- データの解析と問題の認識
- 問題の解決
- 拡張の実装

まとめ

問題の解決

最適化デザインレベル

全般的なチューニング方法論は、システムレベルから始まり、マイクロアーキテクチャー・レベルまで網羅する。特定のチューニング・ゴールに関係なく、高いレベルから低いレベルまで、順に沿って、レベルごとに解析を行なう

- システムレベル
- アプリケーション・レベル
- マイクロアーキテクチャー・レベル

最短時間で最良のスピードアップが得られるようにするには、これらのステップに沿うことが重要

問題の解決

チューニング・ゴールと調査する領域

順序	チューニング・レベル	ゴール	調査する主な領域	スピードアップの可能性
1	高: システムレベル	アプリケーションとシステムの相互作用を改善することにより、アプリケーションをスピードアップする	•ネットワークの問題 •ディスクのパフォーマンス •メモリーの使用量	3 倍
2	中: アプリケーション・レベル	アプリケーションのアルゴリズムを改善することにより、アプリケーションをスピードアップする	•ロック •ヒープの競合 •スレッド化アルゴリズム •API の使用量	2 倍
3	低: マイクロアーキテクチャー・レベル	プロセッサー上でのアプリケーションの実行を改善することにより、アプリケーションをスピードアップする	•アーキテクチャーのコーディング上の問題点 •データ/コードの局所性 (キャッシュ) •データのアラインメント	1.1 ～ 1.5 倍

インテル® VTune™ Amplifier XEを使用して、これらのチューニング方法論を実装し、目標とするプラットフォームで最短時間で最良のスピードアップを実現する

問題の解決

システム・レベル・チューニング

システムリソースの利用方法を最適化し、システムとの相互作用を改善することでアプリケーションのスピードアップを目指す。システム・レベル・チューニングではたいていの場合、最少の作業で最大のスピードアップが可能

1. カウンターモニター・データ・コレクターで OS のパフォーマンス・カウンターを監視
2. システムレベルのパフォーマンス・ボトルネックとシステムの相互作用に注目
3. 改善の余地がありそうな範囲にズームし、チューニング・サマリーに注目

システムレベルのパフォーマンスの問題点をすべて解決した場合、または重大な問題点がないと判断した場合は、チューニングの次のレベルである、アプリケーション・レベル・チューニングに進む

アプリケーション・レベル・チューニング

アプリケーション・レベル・チューニングのゴールは、アプリケーションのアルゴリズムの改善、実装のスレッド化、および API やプリミティブの使用によって、アプリケーションをスピードアップすること

アプリケーション・レベル・チューニングでは、2 つの主要なチューニング戦略を選択できる

- スレッドモデルの向上
- 演算の効率性の向上

問題の解決

スレッドモデルの向上

効率的なスレッドモデルを使用することは、パフォーマンスのスケールを向上するためには非常に重要

シングルスレッド・アプリケーションの場合、パフォーマンスを向上するための最初のステップは、アプリケーションにマルチスレッド機能を追加すること

既にアプリケーションがマルチスレッド化されている場合は、マルチプロセッサー・システム (およびハイパー・スレッディング・テクノロジー搭載の単一プロセッサー・システム) 上でパフォーマンスおよびスケールを向上させる

問題の解決

演算の効率性の向上

サンプリングを使用し、アプリケーションのパフォーマンスに与える影響が高いコード領域を識別する

- ホットスポットの使用
- コンカレンシーの使用
- ロック&ウェイトの使用
- EBS（イベント・ベース・サンプリング）の使用

アプリケーション・レベルのパフォーマンスの問題点をすべて解決した場合、または重大な問題点がないと判断した場合は、チューニングの次のレベルである、マイクロアーキテクチャー・レベル・チューニングに進む

問題の解決

マイクロアーキテクチャー・レベル・チューニング

- 特定のプロセッサー上でのアプリケーションの実行を改善し、アプリケーションのパフォーマンスのスピードアップを目指す
- このチューニング・タイプは、特にプロセッサー集中型アプリケーションに適切
- 開発しているアプリケーションが、プロセッサー集中型ではない場合、マイクロアーキテクチャー・レベル・チューニングを行う前に、システムレベルおよびアプリケーション・レベル ・チューニングを行う

マイクロアーキテクチャー・レベル・チューニングの一般的な方法論:

1. パフォーマンスに与える影響が大きい、最も時間を費やしているコード領域を見つけます
2. インテル® アーキテクチャー上でそれらのコード領域の実行を解析します
3. マイクロアーキテクチャー・レベルでのパフォーマンスの問題点を識別します
4. パフォーマンスを向上するために問題点を回避する方法を決定します

問題の解決

コード・チューニングのレベル

アセンブラー

組み込み関数

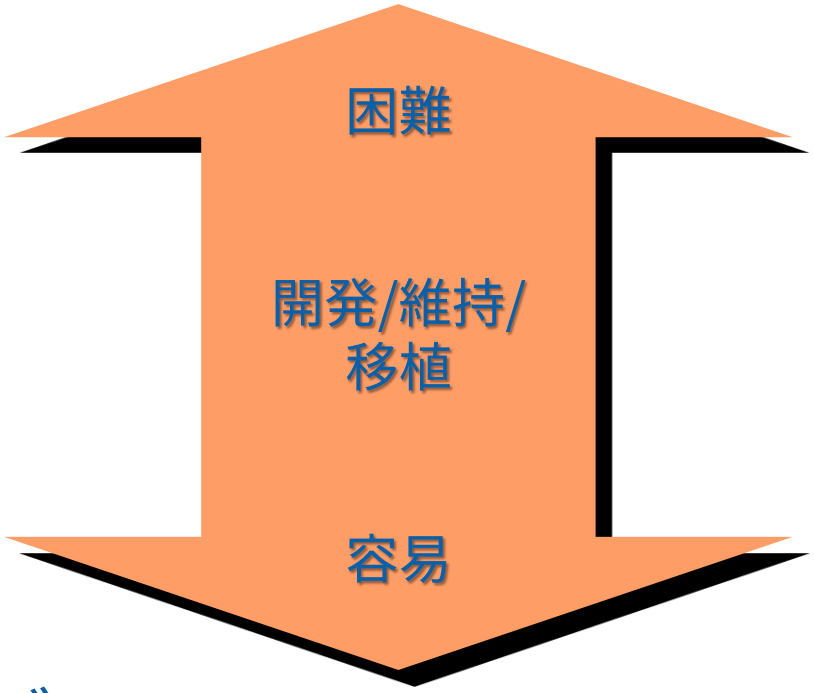
C++ ベクトルクラス

マルチスレッド

ループ変換

インテル® コンパイラー

インテル® パフォーマンス・ライブラリー



問題の解決

コード・チューニング

並列処理の場合

- クラスター別に処理を分散（分散メモリー）
- 単一ノードの最適化
- プロセッサー（コア）別に処理を分散（SMP）

拡張の実装

- インテル® パフォーマンス・ライブラリーを使用する
- 各種コンパイラー・オプションを使用する
- 実装を行う前に、コンパイラーやハードウェアが拡張を自動的に実装するかどうかを調査
- ソースを修正する
  （ループ変換、キャッシュ最適化、SIMD、OpenMP*、組み込み関数、アセンブリー）

テスト

- アプリケーションが拡張後も正しく実行することを確認する(リグレッション・テスト)
- 拡張が実際にパフォーマンスを向上させていることを確認する
- スピードアップを計算する
- 最適化を終了するかどうかを決定する

コースの内容

パフォーマンス・サイクルの概要

パフォーマンス・サイクルの詳細

- パフォーマンス・データの収集
- データの解析と問題の認識
- 問題の解決
- 拡張の実装

まとめ

まとめ

最適化タスク:

- パフォーマンス・データの収集
- データの解析と問題の認識
- 問題の解決
- 拡張の実装
- 結果のテスト

プロセスの各ステップでインテル® ソフトウェア開発ツールを使用する

このセッションを理解するのに役立つ資料

- マイクロアーキテクチャーのトップダウン解析法を使用してアプリケーションをチューニングする
- インテル® VTune™ Amplifier XE の General Exploration (一般解析) がどのように動作するかを理解する
- インテル® HD グラフィックスとインテル® Iris™ グラフィックスを使用するアプリケーションの解析
- チューニング・ガイドとパフォーマンス解析

補足資料

参考資料(英語)

Dowd, Kevin および Charles Severance 著
High Performance Computing. O'Reilly, 1993.

Pasquale, Armenise 著
"A Structured Approach to Program Optimization." *IEEE Trans on Software Engineering.* Feb (1989) 101-108

Bentley, Jon 著
Programming Pearls（第 2 版）Addison-Wesley: 2000

ベンチマーク

- Linpack* ベンチマーク
- NAS
- STREAM
- SPEC* (Standard Performance Evaluation Corporation)
- TPC*

パフォーマンス・サイクルの詳細

スピードアップ

2 つの基本式

$$\text{スピードアップ} = \frac{\text{基準となる時間}}{\text{最適化された時間}}$$

$$\text{スピードアップ} = \frac{\text{最適化されたスループット}}{\text{基準となるスループット}}$$

Big O 記法

"Big O" f(n)=O(g(n)):

- 問題を解決するためのループ中の反復数のスケーリング

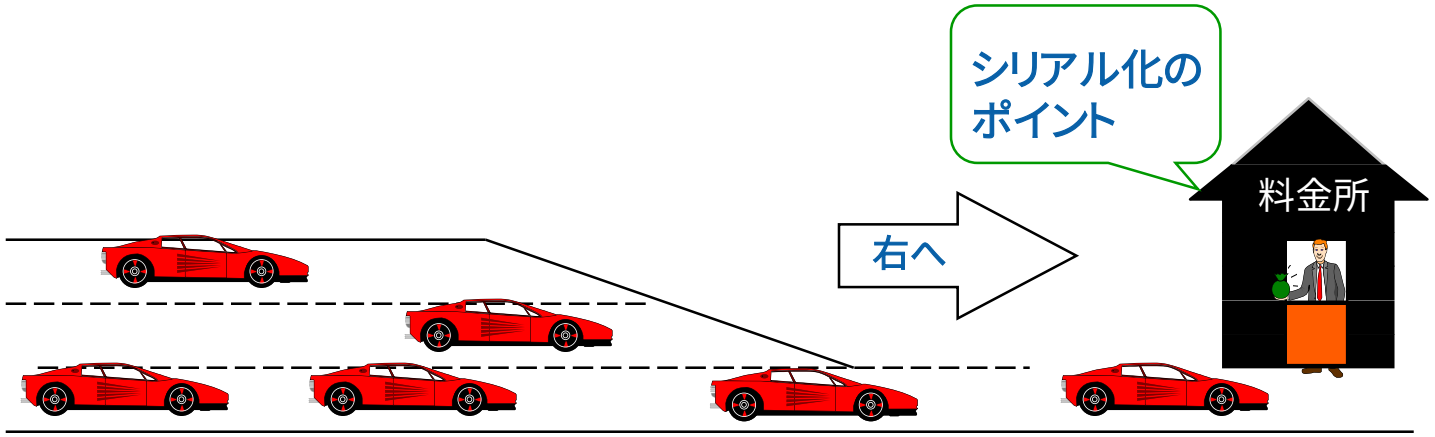
例:

- リニアー検索: O(n)
- バイナリー検索: O(log n)
- バブルソート: $O(n^2)$
- マージソート: O(n log n)

O の判定

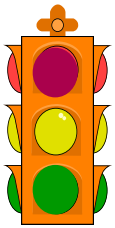
- N=10、100、1000 でループを実行し、時間をプロットして、時間がリニアーに増加しているかそうでないかを確認する

ボトルネックの識別



- ボトルネックは、システムで最も低速な部分である
- ある処理が他の処理が完了するまで待たなければならない場合、シリアル化のポイントが存在する
- ボトルネックは、システムが単位時間あたりに処理できる作業量を最終的に決定する

使用率



リクエストに応じるためにデバイスが稼動していた時間の割合

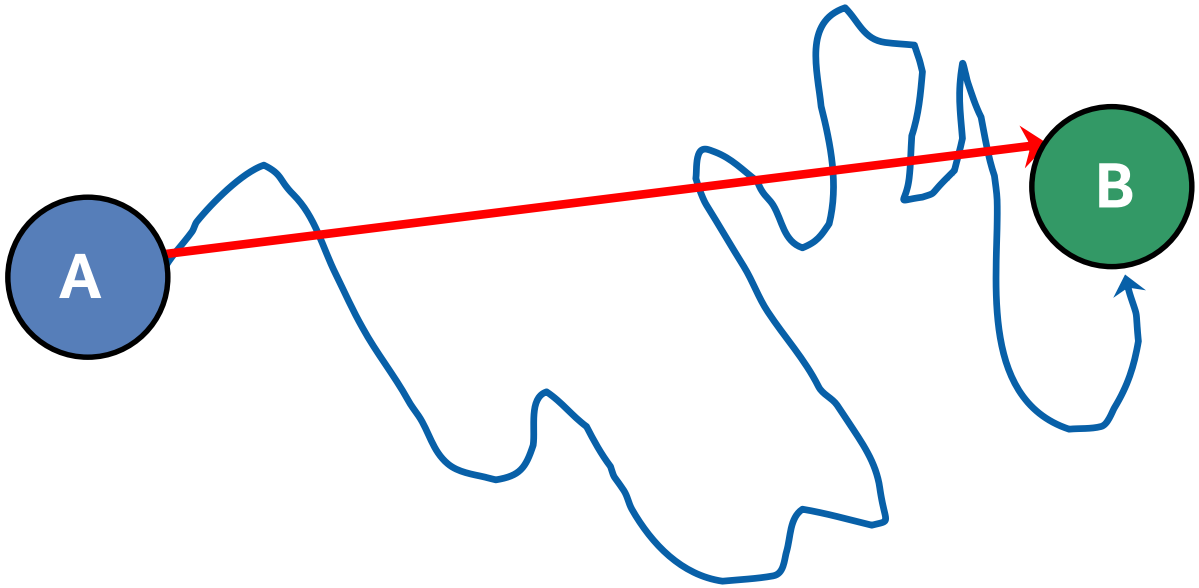
$$\text{使用率} = \frac{\textbf{デバイス稼動時間}}{\textbf{合計時間}} * 100$$

時間の残りの割合は、*アイドル時間*

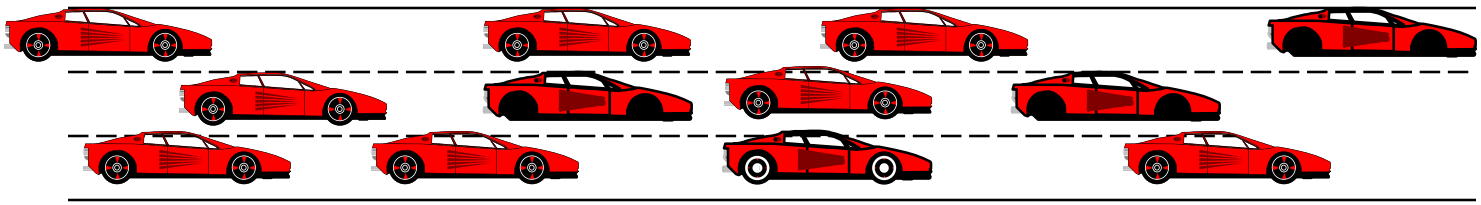
アイドル時間 = (100 - 使用率)

効率

- 必要な作業を行うために実際に費やされた時間の割合のこと
- リクエストに応じている間に、多くのオーバーヘッドやその他の非効率な処理が発生する場合がある
- 使用率が高いことは、必ずしも効率が良いことを意味しない

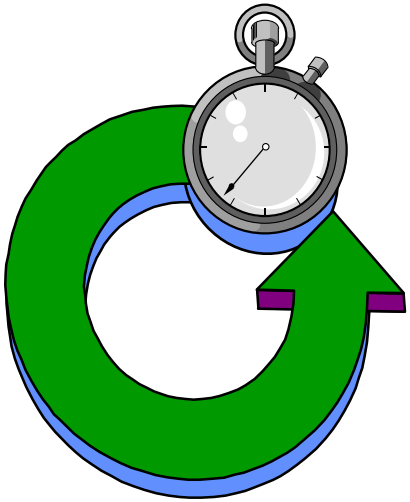


スループット



- 時間単位あたりにシステムが完了できる作業量
- 通常は、上記の値がスループットの上限になる
- 帯域幅は、スループットの１つの測定単位である

レイテンシー



- アクションを完了するために必要な合計時間のこと
- レイテンシーは、サブタスクのいくつかのレイテンシーの累計として表示される
- レイテンシーは、応答時間（レスポンス時間）とも呼ばれる

典型的なボトルネック

メモリー

- 合計メモリー使用率を計算する
- スループットを計算する

I/O

- ディスク上のデータを計算する
- スループットを計算する

CPU

MIPS/FLOPS/CPI

MIPS

- 1 秒あたりの命令数（百万単位）

FLOPS

- 1 秒あたりの浮動小数点演算数

CPI

- 命令あたりのサイクル数

反復ごとのマシンレベル・ストアとロード

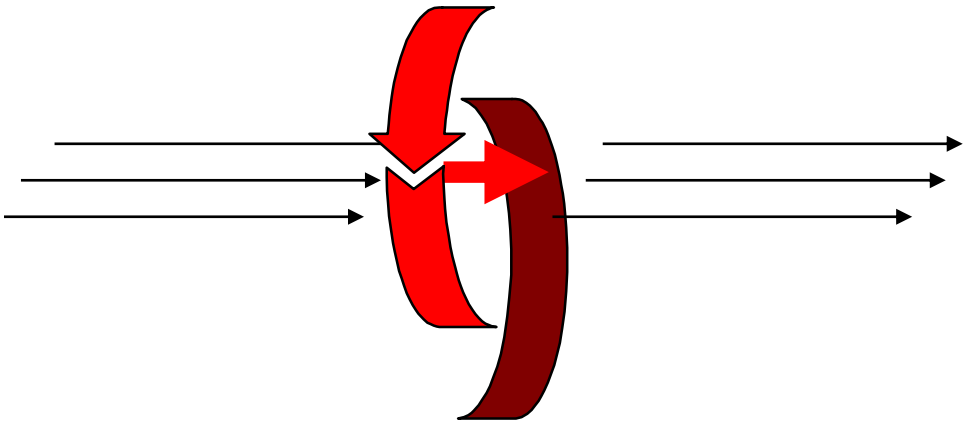
ループ反復ごと:

- メモリーストアとロードの回数（M）をカウントする
- 計算の回数（C）をカウントする

メモリーの帯域幅がボトルネックの場合、C/M を増やす

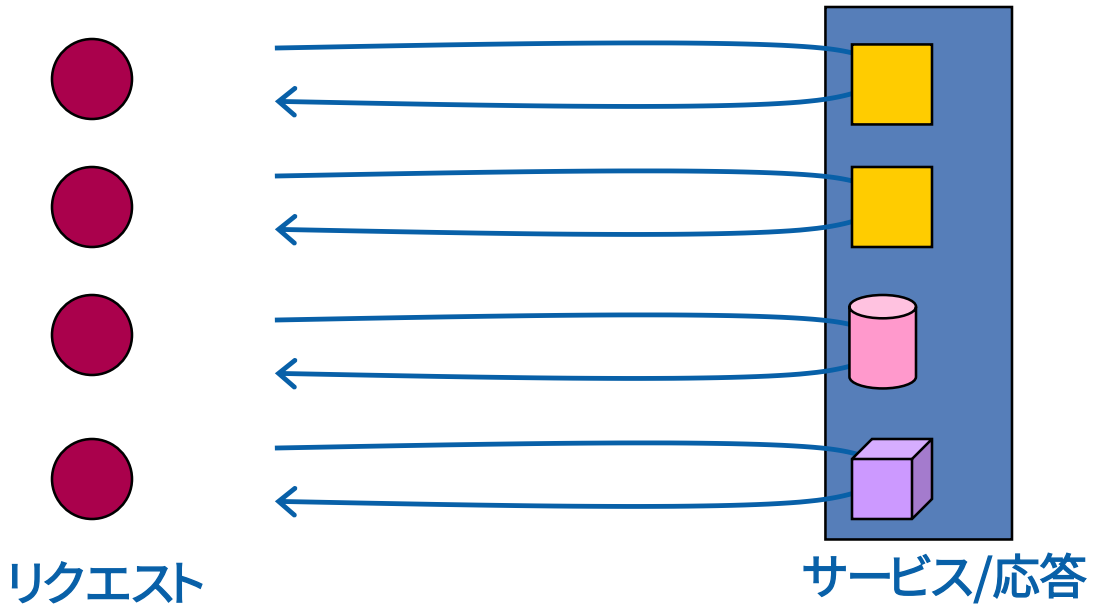
CPU のアクティビティーがボトルネックの場合、C/M を減らす

並行化



- 複数の処理（スレッド、実行形式、その他）を同時に完了できるようにすること
- 並行化は、有効なレイテンシーを減らし、スループットの合計を増やすために使用される

並列化

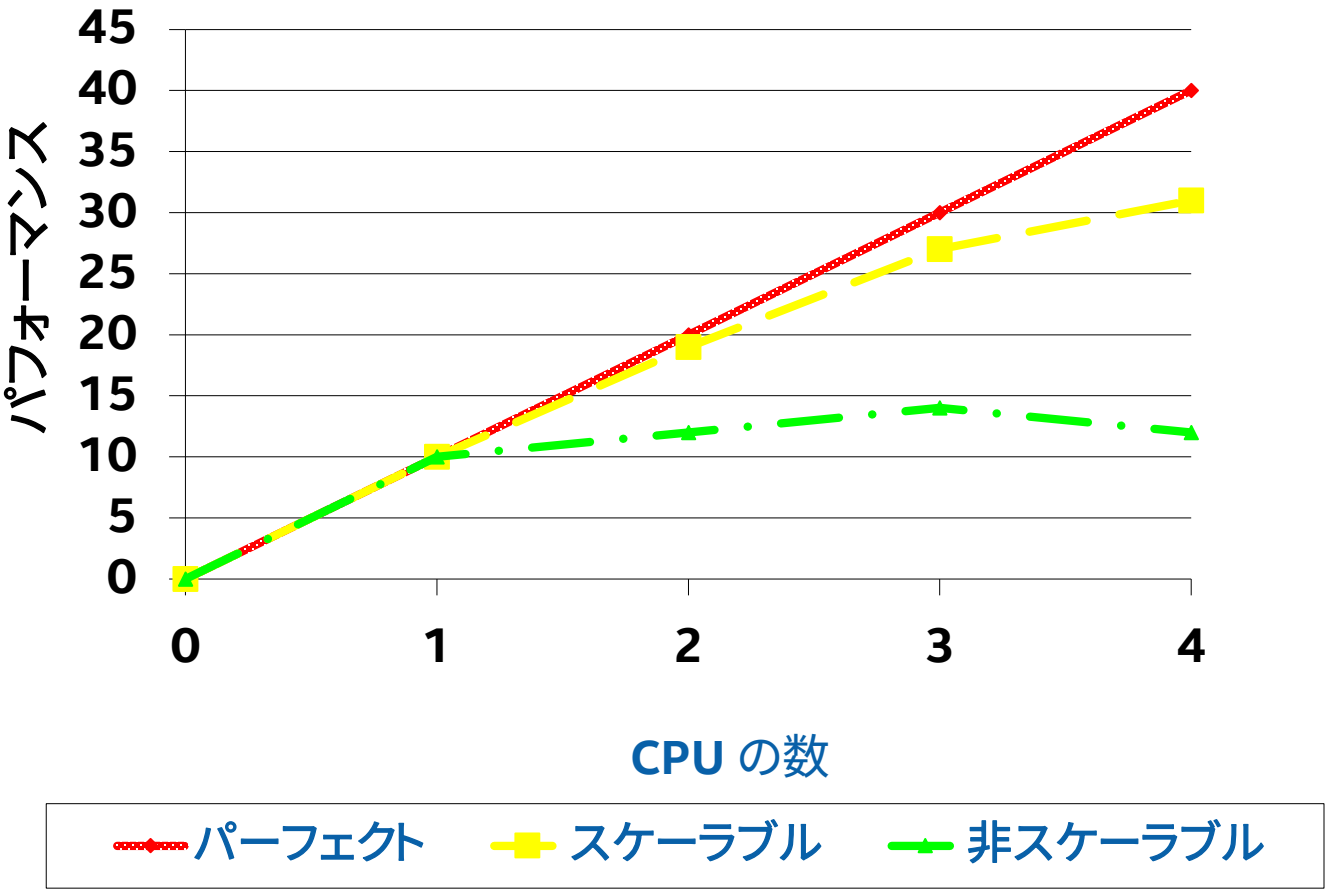


- 単一システムで複数のプロセスまたはプログラムを実行する機能のこと
- 複数のプロセッサーがあるシステムで並列化を行うと、スループットが向上する

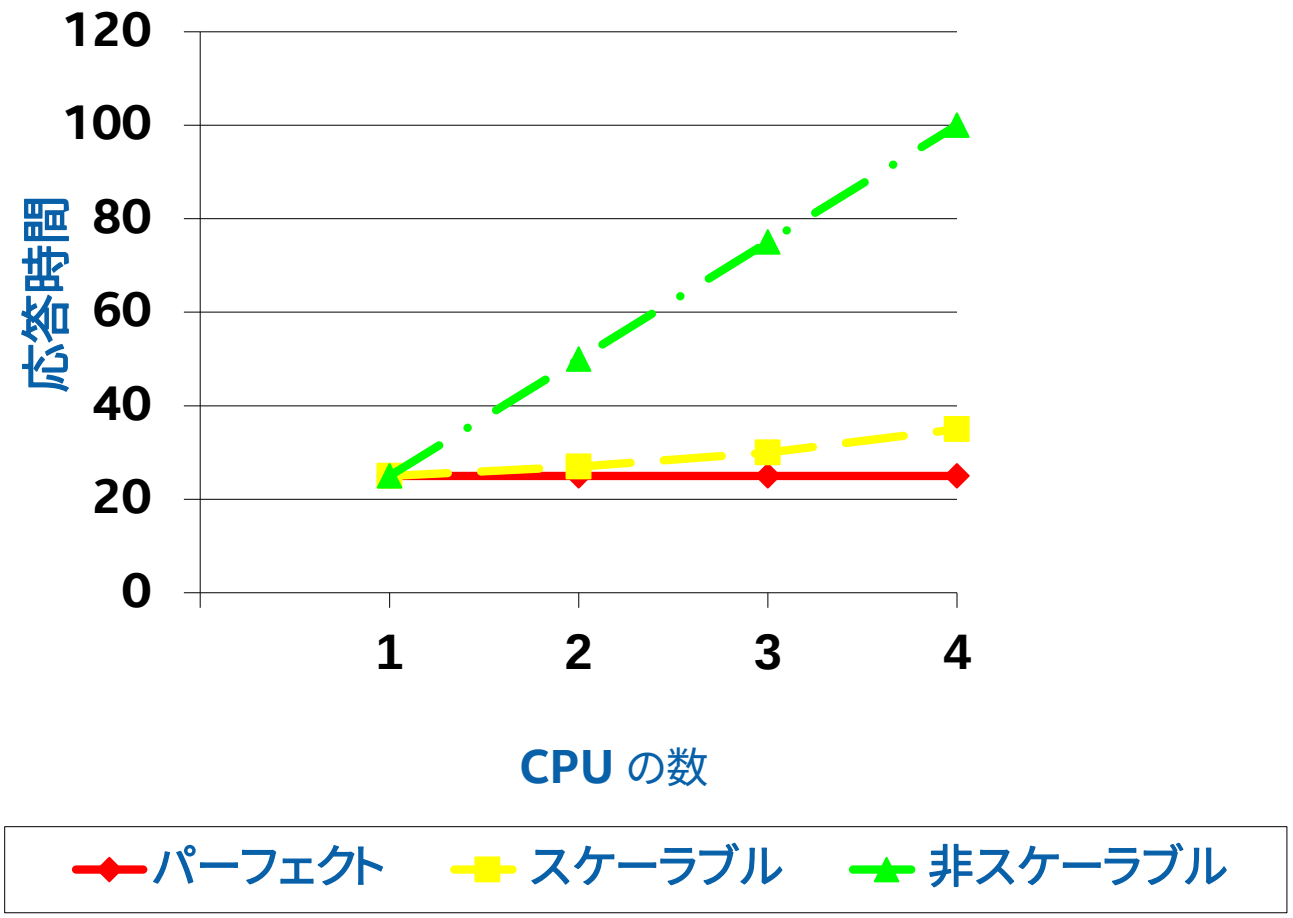
スケーラビリティー

- リソース（つまり、CPU）の数を増やしてパフォーマンスを向上させる機能のこと
- 優れたスケーラビリティーを得るには、一般的に複数の CPU で構成された SMP システムを使用する
- 2 つのメトリクス:
  - 固定ワークロードの場合、CPU の追加によるパフォーマンスの相対的な向上
  - スケーラブルなワークロードの場合、ワークロードと CPU の追加によりパフォーマンスを維持する機能

固定ワークロードのスケーラビリティー



スケーラブルなワークロードのスケーラビリティー



パフォーマンス・データの収集

タイマー

- パフォーマンスに影響しやすいコードの識別
- 便利なタイマー・プロパティー
  - 正確な時間情報の取得
  - アクセス・オーバーヘッドが少ない
  - リセットがほとんど発生しない
- インテル® コンパイラーのプラグマを使用してプロセッサーの内部クロックカウンターにアクセス

アムダールの法則

アムダールの法則の式

$$\text{スピードアップ} = \frac{1}{(1 - \text{並列化率}) + \frac{\text{並列化率}}{\text{並列化によるスピードアップ}}}$$

リトルの法則

タスクの数 = 到着レート x 応答時間

- 到着レートは、システムにリクエストが到着する割合を示す
  - システムから出発したリクエストの割合が、システムに到着したリクエストの割合よりも高い場合、システムは安定している（平衡または定常状態）と言える
- 応答時間には、サービスを待つために費やされた時間と、サービスを受けるために費やされた時間が含まれる

リトルの法則

システムが平衡な場合、タスクの数、到着レートおよび応答時間には、次のような関係がある

サービスキュー内のタスクの数 = 到着時間 x 応答時間

タスクの数　= システム中のタスクの数
到着レート　= タスクが到着する割合
応答時間　= タスクを完了するための平均時間

最適化に関する注意事項

インテル® コンパイラーは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。これには、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。インテル® マイクロアーキテクチャーに非固有の特定の最適化は、インテル製マイクロプロセッサー向けに予約されています。この注意事項の適用対象である特定の命令セットの詳細は、該当する製品のユーザー・リファレンス・ガイドを参照してください。

改訂 #20110804

iSUS