Android で 並行プログラミング

マルチコアを生かそう

Android で 並行プログラミング

robo

Android で 並行プログラミング

- 名　　前: robo（兼高理恵）
- お 仕 事: Java 技術者
  設計から実装まで
- 好きな物: モバイル端末

並行と並列について

並行(concurrent)と並列(parallel)の違い

並行も並列もマルチスレッド・プログラミングです。
どちらも複数のスレッドが実行開始から終了の間で、
同時に存在する(実行中である)ことを表します。

違いは、実行時のコア数(同時スレッド処理CPU数)です。
並行は、(時分割などで)実行時のコア数を問いませんが、
並列は、複数コアでの相互干渉実行を前提としています。

注意:上記は、(厳密でない)簡略化した概念説明です。

Android 端末のマルチコア化

Android 端末のハイパワー化は止まるところを知らず、
ついにマルチコアが当然の時代となりました。

参考先

GALAXY nexus イメージ元
http://www.google.co.jp/nexus/

HTC One X イメージ元
http://www.htc.com/www/smartphones/htc-one-x/#specs

Android 端末のマルチコア化

注意

マルチコアの恩恵が受けられるのは、
Android 3.0以降だそうです。

Android 3.0より、
Dalvik VM や Bionic ライブラリにマルチコア対応が加えられ、
シングルスレッドのアプリであってもガベージコレクションを
空コアで行うなどして、パフォーマンス向上を自動的に行って
くれるそうです。

参考先

モバイルトレンド 塩田紳二　マルチコアがモバイルを救う?（第159回）
http://pc.nikkeibp.co.jp/article/column/20110309/1030674/

Android 3.0 Platform Highlights　Support for multicore processor architectures
http://developer.android.com/intl/ja/sdk/android-3.0-highlights.html

マルチコア化のメリットとデメリット

メリット

- 処理速度の高速化
  適切なコア割り当てとクロック管理がされている場合、
  マルチコアでの並列実行による、処理時間短縮が
  望めます。

- 消費電力の低減化
  適切なコア割り当てとクロック管理がされている場合、
  低クロックでも処理を賄えるため、消費電力低下が
  望めます。

マルチコア化のメリットとデメリット

デメリット

- ソフトウェア・コストの増大
  マルチコアを生かすためのソフトウェア対応が必要
- アプリ実行環境のマルチコア対応
  OSでのスレッド・スケジューリングやクロック制御などの
  マルチコア対応と、アプリケーション・フレームワークでの
  並行プログラミング・サポートも必要
- アプリでのマルチコア対応
  マルチコアを生かすためには、並行(並列)プログラミングが
  必要だが、タスク分割分析や排他制御が必要になるため、
  逐次的プログラミングよりも困難

マルチコア化による処理速度の高速化

アムダールの法則

アムダールの法則(Amdahl's law)
並列実行による高速化率を求める式です。

(※)別の法則もあります

性能向上 P = 1/(F+(1-F)/N)
F : 並列化できない部分の割合 (直列処理率)
N : コア数

マルチコア化による処理速度の高速化

アムダールの法則

並列実行による処理の高速化は、処理全体で直列実行される
(並列実行できない)処理割合の逆数分までが限度です。

実際には、マルチスレッド管理のオーバヘッドもあるので、
論理限界にも至りません。

式より、
高い並列性能を出すには、
直列処理率を減らす=>並列化率を上げる
アルゴリズムが重要となります。

補足:オーバヘッドにより、コア数を増加させても、
ある時点で性能が下がり始めます。

マルチコア化による処理速度の高速化

グスタフソンの法則

グスタフソンの法則 (Gustafson's law)
十分に大きな規模の問題は、効率的に並列化して解くことができる事を示す法則です。

高い並列性能を出すために、
アルゴリズムでなく問題のサイズが重要な場合もあります。

例えばプログラムが、
処理量が問題データ量に依存しない初期化部(直列処理)と、
問題データ量に依存するデータ解析部(並列処理)に分かれる等、
直列実行の時間割合が、問題のサイズに左右される場合があることに留意ください。

マルチコア化による処理速度の高速化

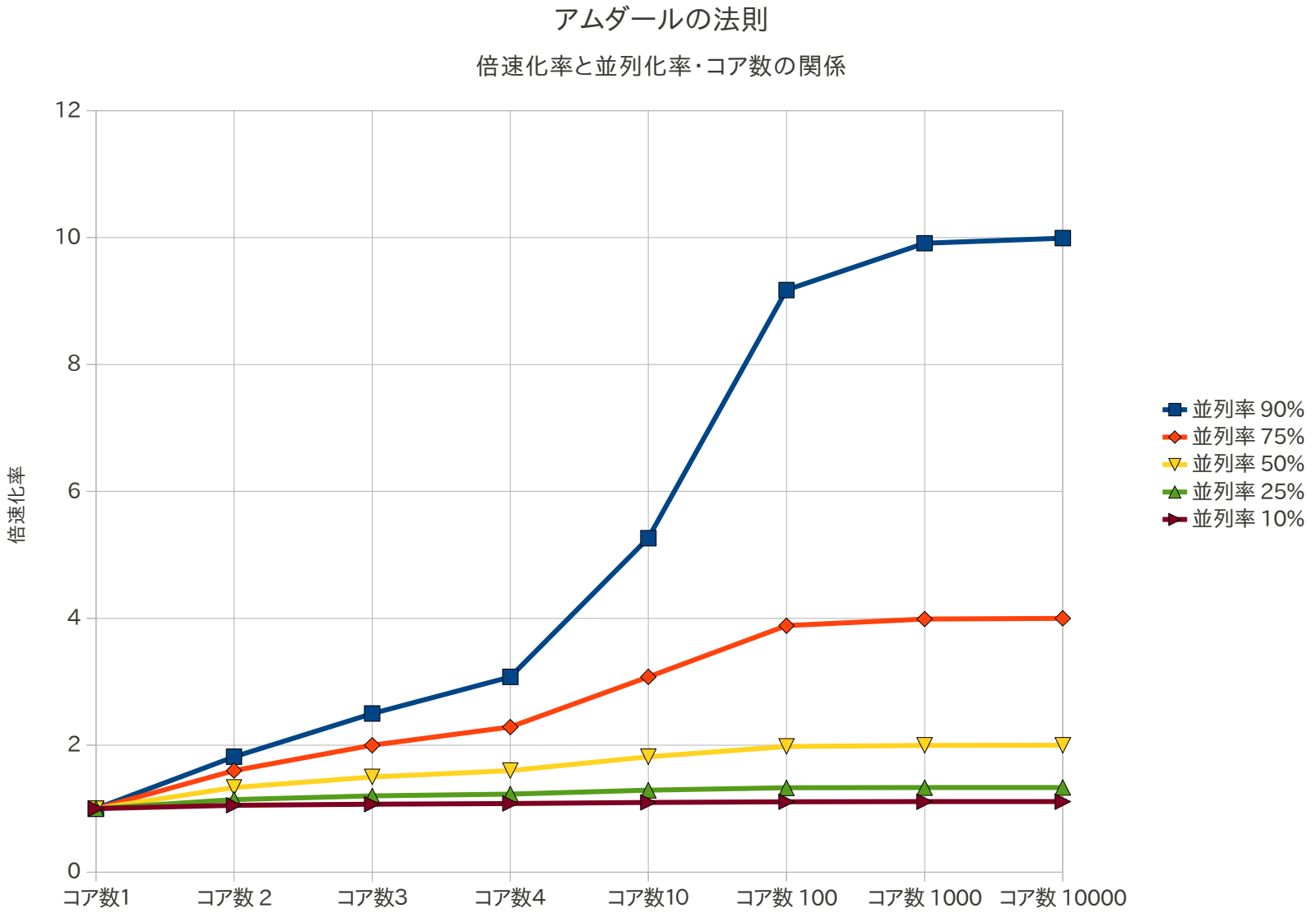
並列高速化の限界

並列高速化の限界
オーバヘッドがなくコア数が無限大としても、
全体処理時間は、直列実行部の処理時間より短くなりません。
例えば、処理全体で25%が直列実行されるプログラムでは、
いくらコア数を増やしても、1/(25% → 0.25)=4 から、
高速化は4倍が限度です。

直列処理率が50%を越えるのなら、
いくらコア数があったとして倍速も望めません。

並列化率90%のプログラムでも、
100台のマシン(コア)で性能飽和してしまいます。

マルチコア化による処理速度の高速化



マルチコア化による処理速度の高速化

参考先

Java 並行処理プログラミング
ソフトバンク クリエイティブ株式会社 ISBN4-7973-3720-6
第11章 実効性能とスケーラビリティ 11-2章 アムダールの法則

並行コンピューティング技法
オライリー・ジャパン ISBN978-4-87311-435-4
3章 正当性の検証と性能測定 3.3.1.1 Amdahlの法則 3.3.1.2 Gustufson-Barsisの法則

Wikipedia アムダールの法則
http://ja.wikipedia.org/wiki/%E3%82%A2%E3%83%A0%E3%83%80%E3%83%BC%E3%83%AB%E3%81%AE%E6%B3%95%E5%89%87

Wikipedia グスタフソンの法則
http://ja.wikipedia.org/wiki/%E3%82%B0%E3%82%B9%E3%82%BF%E3%83%95%E3%82%BD%E3%83%B3%E3%81%AE%E6%B3%95%E5%89%87

性能向上は難しいーアムダールの法則
http://blogs.wankuma.com/episteme/archive/2010/01/06/184680.aspx

@IT MONOist
スパコン「京」を使った設計が無料で試せる日がくる？（1/3）
http://monoist.atmarkit.co.jp/mn/articles/1112/14/news011.html

マルチコア化による消費電力の低減化

消費電力計算式

wikipedia「並列計算」より引用

マイクロプロセッサの消費電力は
P=C×V^2×F という式で与えられる。

P は消費電力、
C はクロックサイクル毎に切り替えられる静電容量
(入力が変化するトランジスタの総数に比例)、V は電圧、
F はプロセッサの周波数(正確には1秒あたりのサイクル数)である。

従って、
クロック周波的数が高くなると、プロセッサの消費電力も増大する。

一般的にクロック周波数が高くなると電圧も高くなることから、
クロック周波数が低くなれば、消費電力は相乗で下がります。

マルチコア化による消費電力の低減化

メーカの消費電力低下主張

nVIDIA ホワイトペーパー
「モバイル機器における マルチコア CPU のメリット」より抜粋
nVIDIA Tegra 2 (デュアルコア ARM Cortex-A9)の広告資料

シングルコア CPU で消費される電力を「P」としましょう。
また、CPU の動作電圧は 1.1V、動作周波数は 1GHz だとしましょう。

同じタスクを、NVIDIA Tegra ア-キテクチャ-採用の
デュアルコア Cortex-A9 CPU で行ったらどうなるでしょうか。

～長いので以降は、意訳～
Cortex-A9 は、1Ghz デュアルコアのため、コア利用率が50%ですみ、
負荷が分散低下されているので、動作電圧と動作周波数も下がります。
コアふたつとも、0.8V 550MHz になったとすれば、
シングルコアに比べて、60% しか電力消費しないと主張しています。

マルチコア化による消費電力の低減化

メーカの消費電力低下主張

nVIDEAの主張を検証してみると、
下記の計算式では、たしかに2コア合算しても60%になります。

周波数毎の静電容量が不明のため、全て 1 と仮定しています。
シングルコアCPU:1 x 1000(mhz) x 1.1^2(v) = 1210(単位不明)
Cortex-A9 CPU:1 x 550(mhz) x 0.8^2(v) = 352(単位不明)
=> 2コアなので2倍にする 352 x 2 = 704(単位不明)
消費電力割合: 702 / 1210 x 100= 58.181818% => 約60%

Android システムからのプロセッサ周波数制御手法が不明なので
正しくないかも知れませんが、
一度に処理するデータ量とスループットが適度に負荷分散されるなら、
低いCPUクロックで定常化して処理が実行され、
低消費電力が実現されるのでしょう。(クロック数監視とジョブコントロールが必要)

マルチコア化による消費電力の低減化

クロック周波数取得方法について

動作クロック周波数の確認

Linuxカーネルには、CPUが持っている動的なクロック周波数の変更機能を利用するための CPUfreq という仕組みがあり、

カーネルの cpufreq ドライバにより、動的にCPUクロック速度のスケーリングを行っているそうです。

ただし、GALAXY nexus で確認した限りですが、
cpufreq の構成がデスクトップと異なります。
詳しい情報を得ていませんので、この情報は、
参考程度に止めてください。

マルチコア化による消費電力の低減化

クロック周波数取得方法について

クロック周波数確認 *(GN は、デュアルコアなのでパス cpu0 と cpu1 が参照可能)*

最大値:例 1200000 (KHz)
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq

最小値:例 350000 (KHz)
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq

現在値:例 (*rootでないと確認不能*)
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq

スケール可能値:例 350000 700000 920000 1200000
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies

スケール現在値:例350000
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq

(注)governor と cpufreaq コアによって決定された
　現在のCPU周波数の KHz (カーネル想定のCPU実行周波数)

マルチコア化による消費電力の低減化

参考先

Android Zaurusの日記　Transformer Primeのマルチコアとクロック
http://d.hatena.ne.jp/androidzaurus/20120127/1327653181

似非玉人の日常　Android 実機のCPUクロック周波数を取得するには？
http://ameblo.jp/zzr250/entry-10820600365.html

試験運用中なLinux備忘録
動的にCPUクロックや電圧を変更するcpufreqの概要とcpufreqdデーモンについて
http://d.hatena.ne.jp/kakurasan/20070720/p1

fedora に関する文書　3.2. CPUfreq Governors の使用
http://docs.fedoraproject.org/ja-JP/Fedora/15/html/Power_Management_Guide/cpufreq_governors.html

TEXAS INSTRUMENTS OMAP-L1 Linux Drivers Usage (5 Dec 2011)
http://processors.wiki.ti.com/index.php/OMAP-L1_Linux_Drivers_Usage

CPUFreq User's Guide
http://lxr.linux.no/linux+v2.6.32/Documentation/cpu-freq/user-guide.txt

アプリ実行環境のマルチコア対応

PC環境でのソフトウェア対応

デスクトップPC環境では、
OSのマルチコア対応だけでなく、
並行(並列)プログラミングの負担を下げるため、
LinuxなどのOSには、並行(並列)実行用のライブラリ(*1)が、
Java7 では、細粒度タスク実行用のフレームワーク(*2)が、
公開/提供されています。

(*1)OpenMP、OpenCL、Pthread
(*2)Fork/Joinフレームワーク

アプリ実行環境のマルチコア対応

Linux OS

Linux では、kernel 2.6 で
マルチプロセッサー向けの改良が行われ、
スケジューラーとラン・キューという仕組みにより、
マルチコア対応済みです。

参考先

@sekitoba「はじめてみよう、プラットフォーム！kernel 2.6のマルチコア対応」(資料)
https://docs.google.com/present/view?id=0AbUQ4TSuB39EZGNqMjUzMnpfMTlkdjQ0Y3NmeA&hl=en&authkey=COm0qqED&pli=1

Yokohama Android Platform Club　第06回勉強会
http://www.yokohama.android-pf.org/study/20110211

アプリ実行環境のマルチコア対応

OpenMP

OpenMP は、
主に共有メモリ型並列計算機で用いられる、
明示的なスレッドの生成や実行と同期を隠蔽化し、
スレッド制御を抽象化するスレッドライブラリです。

C/C++では、pragma omp … 宣言だけで利用でき、
高水準であるため、処理の並行化指定も簡易ですが、
明示的なスレッド操作よりも表現力が制限されます。

アプリ実行環境のマルチコア対応

OpenMP

OpenMP は、fork-join モデルで並列実行を管理します。
大まかに説明すれば、ソースコード中の並行実行可能部分
(パラレルリージョン/parallel region)の範囲を明示させ、
パラレルリージョンのスレッド割り当てや起動(fork)と実行、
全実行完了の待ち合わせ(join)と完了後のメインスレッド
再開など、

パラレルリージョンの同期処理と実行順序の一貫性を
保証します。

アプリ実行環境のマルチコア対応

OpenMP

補足:
パラレルリージョン外の変数は、デフォルトで共有変数となる他、
内部の変数も、任意にスレッドローカルか共有かを指定でき、
共有リソースの同期プログラム負荷も低減してくれます。

Linux用のGNU GCCコンパイラは、OpenMPに対応していますが、
無償で利用できる開発環境は、あまり公開されていないそうです。

参考先

並行コンピューティング技法　オライリー・ジャパン　ISBN978-4-87311-435-4
5章 スレッドライブラリ　5.1 抽象化ライブラリ 5.1.1 OpenMP

マルチコアCPUのための 並列プログラミング 並列処理＆マルチスレッド入門
秀和システム　ISBN4-7980-1462-1
6章 より進んだ並列処理の世界 6.1 OpenMP:標準APIによる共有メモリ型並列プログラミング

Wikipedia OpenMP
http://ja.wikipedia.org/wiki/OpenMP

アプリ実行環境のマルチコア対応

Pthread

Pthread(POSIX thread)スレッドライブラリは、
Linux で利用できる、
スレッドの生成や実行、およびスレッドと共有リソースの同期や
連携を管理する明示的スレッドライブラリです。

参考先

GNU/Linux でのスレッドプログラミング *Linux でのスレッドプログラミング(Pthreadライブラリの使い方)について解説されています。*
http://www.tsoftware.jp/nptl/

並行コンピューティング技法
オライリー・ジャパン ISBN978-4-87311-435-4
5章 スレッドライブラリ 5.2 明示的スレッドライブラリ 5.2.1 Pthread

アプリ実行環境のマルチコア対応

OpenCL

OpenCL は、
GPU(graphics processing unit)に特化した
並列コンピューティングのスレッドライブラリです。

Wikipedia「OpenCL」より引用

> OpenCL(オープンシーエル、Open Computing Language)は、
> OpenCL C言語による、マルチコアCPUやGPU、Cellプロセッサ、
> DSPなどによる異種混在の計算資源(ヘテロジニアス環境、
> Heterogeneous)を利用した
> 並列コンピューティングのための
> クロスプラットフォームなフレームワークである。

参考先
並行コンピューティング技法
オライリー・ジャパン ISBN978-4-87311-435-4
5章 スレッドライブラリ 5.3 その他のスレッドライブラリ

アプリ実行環境のマルチコア対応

Java 並行プログラミング機構

Java 言語における並行プログラミング機構は、時代とともに追加(強化)されていきましたので、その変遷を記述します。

Java 言語における、並行プログラミング機構の変遷
IBM Developer Works Javaの理論と実践: フォークを活用する、第1回
序文「ハードウェアの傾向によってプログラミングのイディオムが変化する」
からの抜粋概要

> *Java 言語には、1995年の誕生時から*
> *synchronized や volatile などの同期プリミティブや、*
> *Thread などのクラスがクラス・ライブラリーに含まれていました。*
>
> *ですが、(マルチプロセッサやマルチコアなどが一般的でない)*
> *1995年当時のハードウェアの現実を反映した機構となっていたので、*
> *スレッドは、並行性ではなく非同期を表現するために*
> *主に使われていました。*

アプリ実行環境のマルチコア対応

Java 並行プログラミング機構

マルチプロセッサー・システムが安くなり、
ハードウェアによる並列処理を活用するアプリケーションが増加すると、
Java 言語のプリミティブやクラス・ライブラリーで提供される
下位レベルの機構を使った並行プログラム作成作業は、
容易ではなく間違いを起こしやすいことが明らかになりました。

Java 5 では、java.util.concurrent パッケージが追加され、
サーバでの大量のユーザー・リクエスト処理など
比較的粒度の粗いタスクを持つプログラムに適した、
並行アプリケーションを構築するための便利なコンポーネント・セットが
提供されました
(並行コレクションやキュー、セマフォー、ラッチ、スレッド・プールと、
Executor フレームワークなど)。

アプリ実行環境のマルチコア対応

Java 並行プログラミング機構

やがてマルチ･コアの時代に入るのに合わせ、
大量のユーザー･リクエスト処理などの粒度の粗いタスクから、
(処理を独立した小タスクに分割する)細粒度の並列処理が
求められるようになりました。

そして、Java 7 には、細粒度並列アルゴリズムを表現する
Fork/Join のフレームワークが追加されました。

参考先

IBM Developer Works Javaの理論と実践: フォークを活用する、第 1 回
序文「ハードウェアの傾向によってプログラミングのイディオムが変化する」からの抜粋概要
http://www.ibm.com/developerworks/jp/java/library/j-jtp11137.html

ちなみに、この記事を書いた Brian Goetz さんは、書籍 Java 並行処理プログラミングの著者(の一人)でもあります。

アプリ実行環境のマルチコア対応

Android環境でのソフトウェア対応

Linux OS の Android 対応 ○

Android 1.6 (IS01)でも、
Kernel version は、2.6.29 です。

アプリ実行環境のマルチコア対応

Android環境でのソフトウェア対応

OpenMP の Android 対応 ✕

OpenMP の Android NDK 対応については、
(現時点2012/02/17で)サポート予定が無いそうです。

参考先

android-ndk グループ　OpenMP support in NDK
http://groups.google.com/group/android-ndk/browse_thread/thread/a547eac5446035b4

David Turnerさんの投稿(2012/02/17)
現時点(2012/02/17)で正式にOpenMP をサポートする予定が無いそうです。 ただし、実験は歓迎するそうです。

アプリ実行環境のマルチコア対応

Android環境でのソフトウェア対応

Pthread の Android 対応 ○

Pthread は、Android ソースの中でも利用されています。

bionic の libcに含まれています。
bionic/libc/bionic/pthread.c
bionic/libc/include/pthread.h

アプリ実行環境のマルチコア対応

Android環境でのソフトウェア対応

OpenCL の Android 対応 ✕

OpenCL は、元々モバイルデバイス用のAPIでなく、
ハードウェア・ベンダの協力も薄いそうです。

現時点の Android で GPU を活用するには、
OpenGL ES2 や RenderScript での描画になります。

参考先

stack overfrow How to use OpenCL on Android? (2012/01/25)
http://stackoverflow.com/questions/9005352/how-to-use-opencl-on-android

アプリ実行環境のマルチコア対応

Android環境でのソフトウェア対応

Java synchronized ○

Java7 Fork/Join ×

ご存知の通り、Android SDK の Java ライブラリには、
Fork/Join フレームワーク・クラスが含まれていません。
他の細粒度並行処理支援フレームワークも提供されていません。

アプリでのマルチコア対応

Android環境でのソフトウェア対応

現時点(2012/04/21)では、
Android はデスクトップ環境から大きく遅れ、
細粒度の並列処理プログラミングを行うための
並列プログラミング開発支援が無いことが判りました。

このため、
細粒度の並列処理プログラムを行うには、
既存の低レベルなスレッド機構(*1)を利用した、
独自のマルチスレッド･プログラミングが必要です。

(*1)NDK での Pthread
　Java での低レベルなスレッドAPIなど

アプリでのマルチコア対応

並列性能アップの万能解はない

高速化と低消費電力化のためには、
いずれも、高い並列性能を出すことが必要です。

独自のマルチスレッド・プログラミングは、
並列化率を上げるアルゴリズムが重要となります。

だだし、アプリの実装内容は様々です。
このため対応アルゴリズムに万能解はありません。

低レベルな並列機構を駆使して、
マルチスレッド・プログラミングの基本を踏まえた、
アルゴリズムを考え地道に作業するしかないでしょう。

Android で並列プログラミングを行うには

Android （現時点）で、
細粒度の並列処理プログラムを行うには、
既存の低レベルなスレッド機構を利用した、
地道な並列化率を上げるアルゴリズムの改良による、
独自のマルチスレッド・プログラミングが必要です。

以降より、マルチスレッド・プログラミングの注意事項を
簡単に紹介します。

短文化した概要説明となっていますので、
詳しくは、各項末の参考資料等で御確認ください。
同期スレッド・プログラミングの経験があることを想定しています。

マルチスレッド･プログラミングの困難性

1. 同時実行処理内容の全組み合わせを考慮しなくてはならない。
   スレッドの実行順序は、スケジューラ(OS側)が決めるため、
   プログラムからは予測できません。

   このため、同時に実行される全スレッドによる処理を把握して、
   全ての実行順序の組み合わせを考える必要があります。
   (このことをインターリーブやインタリーブと呼称します)

   インターリーブでは、共有リソースの整合性確保だけでなく、
   マシンコード(マシン語)レベルでどのような実行がされるか
   認識しておく必要があります。

   逐次処理のプログラムと違い、ソースコードを追うだけでは、
   処理内容を正しく理解できません。　項目の最後に、用語説明をつけました

参考先
IBM DeveloperWorks　double-checked lockingとSingletonパターン（アウトオブオーダー書き込み）
http://www.ibm.com/developerworks/jp/java/library/j-dcl/
ソースコードとマシンコードで、ニュアンスが異なる可能性について記述されています。

マルチスレッド･プログラミングの困難性

2.分析と設計とテストとチューニングの覚悟が必要です。
(1)より、しっかりした処理分析とタスク設計を行わないと、
想定外の実行順序やハードウェア変更(コア数増減)から、
バグやデッドロックに競合と争奪の性能劣化が発生します。

これらは、想定外なのですから設計時には気づけません。
このためテストとチューニングの覚悟が必要になります。

また設計時には、将来のデータ量やコア数の増加に備えて、
あらかじめロードバランシングの考慮や、
スケーラブルなアルゴリズムの採用が必要でしょう。

マルチスレッド･プログラミングの困難性

3. コンパイラによる最適化を意識しなくてはなりません。
マルチスレッド下で非同期に変更されるデータを利用する場合、
コンパイラの最適化によるコード変更も考える必要があります。

同期が高コストだからといって、ロジックで回避したつもりが、
最適化されたコードでは、無意味にされているかもしれません。

マルチスレッド下で、実行順序や値の最新値の参照を
保証したい場合は、
同期を指定して最適化を抑止してください。

マルチスレッド･プログラミングの困難性

単純なサンプル:
以下のような処理の場合、コンパイラは、
watch[0] が永遠に 0 と判断して最適化し、
if(watch[0]!=0) whatWork(); のコードを
作成しないかもしれません。

```
//NDK 処理で非同期に変更されるデータ
byte[] watch = new byte[1];

～処理省略～
watch[0]=0;
while(true){
  ～処理省略～
  if(watch[0]!=0) whatWork();
  ～処理省略～
}
～処理省略～
```

参考先
IBM DeveloperWorks double-checked lockingとSingletonパターン（double-checked locking: その2）
http://www.ibm.com/developerworks/jp/java/library/j-dcl/
最適化によるロジックの無効化について記述されています。

マルチスレッド･プログラミングの困難性

4.プロセッサ(ハード)の特性を意識する必要もあります。

プロセッサ内の内部キャッシュが、キャッシュライン･サイズ毎に
管理されていた場合、異なる変数が同じキャッシュラインに含まれる
状態を偽共有(false sharing)と呼称します。

スレッドAの変数(a)と、スレッドBの変数(b)が同一キャッシュラインに
含まれる場合、プロセッサは、(a)と(b)のキャッシュを同一に扱うので、
(a)の変更により(b)も、(b)の変更により(a)もキャッシュ･フラッシュ
してしまい性能劣化が発生することもあります。

その他、OSに1プロセッサを論理的に複数に見せる(効果を発揮させる)
ハイパースレッディングの原理等を意識する必要もあるでしょう。

参考先

コンピュータアーキテクチャの話　194 キャッシュラインのフォールスシェアリング
http://news.mynavi.jp/column/architecture/194/index.html

.NET の問題　偽共有
http://msdn.microsoft.com/ja-jp/magazine/cc872851.aspx

マルチスレッド･プログラミングの困難性

5. 数学的･計算機科学的な素養を要求されることもあります。
   並列化アルゴリズムだけでなく、アルゴリズム全般で調べ物をすると、
   「*このアルゴリズムは、最悪で O(n log n) の計算量になる*」とか、
   チューニング目安の具体的な実行効率の予測式が例示される
   だけでなく、
   ソートや探索などで、既存のアルゴリズムの知識を前提として、
   全体の説明を省いている解説を見ることもあります。

   やはり、数学や計算機科学の素養(知識)があると良いのでしょう。

マルチスレッド･プログラミングの困難性

6. 並列化できない処理(問題)もあります。
   単純化すれば、あるタスクAが実行時に入力値を必要とするが、
   その入力値が他のタスク処理の結果値であり、かつAの結果が、
   次回のAの入力の元となる繰り返し処理(ループ)を想定すれば、
   個々のループは、並列実行できないことが容易に判ると思います。

マルチスレッド・プログラミングの困難性

参考先

並行コンピューティング技法
実践マルチコア/マルチスレッドプログラミング
オライリー・ジャパン　ISBN978-4-87311-435-4
　1章　早くししたい人、手を挙げて!
　2章　並行か非並行か？　それが問題だ
　3章　正当性の検証と性能測定
　4章　マルチスレッドアプリケーション設計の8つのルール

マルチスレッド･プログラミングについて

一般的な並列化作業の流れ

並列実行のマルチスレッド･プログラムとは、
並列実行させるタスクの集合体です。

任意の要求を果たす逐次処理プログラム(*1)は、
並列実行できるようなタスクに分解してあげなくてはなりません。
(*1)問題解決のアルゴリズム

逐次処理をタスクに分解する万能解法は、ありません。
全処理から、
依存性なく一括実行できる処理単位の抽出と、
依存性のある一括実行処理単位の抽出をしてから、
各種並列化手法用のタスク化を行うことになるでしょう。

マルチスレッド･プログラミングについて

逐次処理をタスクに分解する際の注意事項

- 依存性のない一括実行処理単位同士は、並び替え可能です。
- 依存性のある一括実行処理単位は、
  依存関係のある処理単位同士での同期や逐次実行が必要です。
- 依存は、並列性を低下させます。
  タスク分解中に見つけた共有リソースへの読書や依存関数の呼び出しは、
  減らすか無くすようにするか、同一スレッドからしかアクセスさせないなどの
  工夫が必要です。

一般的には、独立したループ処理(*1)を探し出して(作り出して)、
既知のいくつかの並列化手法に即したタスク化を試みます。

(*1)逆順にループを回しても同じ結果が得られるような、
　　ループ内に依存性がない(と思われる)繰り返し処理

マルチスレッド･プログラミングについて

一般的な並列化手法

データ並列化手法

データを分割して同時処理する手法です。
複数スレッドから読書共有されないデータを分割して、
延べデータ分割個数分のスレッドで、
各々の分割データをタスク集合体に処理させます。

例えば、配列データの各値を2倍化した新配列を生成
するような、ループを回して、入力データのみ異なる
同じ処理を繰り返す場合に有効です。

マルチスレッド･プログラミングについて

一般的な並列化手法

タスク並列化手法

タスクを空スレッドに振り分ける手法です。
タスク集合体を入れるジョブキューを用意して、
実行が必要なタスクをジョブキューに入れていき、

ジョブキューが空になるまで、
スレッド(全体で複数個あります)ごとに、
実行可能となったタスクをキューから取り出して処理し、
処理が終われば、次のタスクを処理してもらいます。

マルチスレッド･プログラミングについて

一般的な並列化手法

パイプライン並列化手法

処理を分割して流れ作業させる手法です。
順番に実行する役割のスレッドを用意しておき、
一連の処理を行う複数のタスク実行を可能なら並び替え、
「依存性のあるタスク同士が求める順番に実行されるよう、
　順序づけしたスレッドに割り当てられるよう」分割して、
流れ作業のように、順序依存のあるタスクを処理させます。

(注)各スレッドが同じ負荷になるよう、負荷分散する必要があります。

マルチスレッド･プログラミングについて

一般的な並列化手法

分割統治法アルゴリズム

問題を分割してから統合する手法です。
問題を適切に解決可能(しきい値)になるまで、
2つ以上の副問題に分割して再帰させることと、
解決可能になれば、次々と処理させて結果を待ち、
全ての結果内容を統合して解を得るアルゴリズムです。

参考先

IBM Developer Works　Javaの理論と実践: フォークを活用する、第 1 回　　分割統治法アルゴリズム
http://www.ibm.com/developerworks/jp/java/library/j-jtp11137.html

マルチスレッド･プログラミングについて

主な参考先

マルチコアCPUのための　並列プログラミング
並列処理＆マルチスレッドプログラミング
秀和システム　ISBN4-7980-1462-1

並行コンピューティング技法
オライリー･ジャパン　ISBN978-4-87311-435-4

Java　並行処理プログラミング
ソフトバンク　クリエイティブ株式会社　ISBN4-7973-3720-6

組み込みマルチコア進化論(1)　マルチコアで処理時間短縮の前にやるべきこと
http://monoist.atmarkit.co.jp/mn/articles/0811/05/news136.html

組み込みマルチコア進化論(2)　マルチコア環境における並列化有効ポイント
http://monoist.atmarkit.co.jp/mn/articles/0812/18/news123.html

組み込みマルチコア進化論(3)　マルチコアで高速化処理を実現するための手法（データ並列化/パイプライン並列化）
http://monoist.atmarkit.co.jp/mn/articles/0901/16/news148.html

組み込みマルチコア進化論(4)　マルチコアの処理単位並列化手法（タスク並列化）
http://monoist.atmarkit.co.jp/mn/articles/0905/07/news087.html

組み込みマルチコア進化論(5)　マルチコアにおける標準表記OpenMP
http://monoist.atmarkit.co.jp/mn/articles/0906/11/news099.html

マルチスレッド･プログラミング用語集

「同期」とは、
　複数スレッドが協調して、全スレッドで統一の何かを行うこと。
　ロックや排他制御による、共有リソースやブロックへの
　同時アクセスを制限した一貫性確保や、全スレッドが実行される
　まで待機させるスレッド･スケジュールの制御などを表します。

「依存性」とは、
　並列化できなくしてしまう、他タスクとの関係 (仕様) を表します。
　例えば、「タスクAは、タスクBの結果を受け取る」なら順序依存を
　「Xは、必ず以前の値より1大きくなる」ならデータ依存を持ちます。

マルチスレッド･プログラミング用語集

「スレッドローカルなデータ」とは、
　他のスレッドからアクセスできないデータを表します。
　スレッド実行中の関数内のローカル変数や、
　スレッド実行中に生成したオブジェクトは、
　スレッド毎のスタックに確保されるのでスレッドローカルです。

「ロードバランシング」とは、
　負荷分散とも呼ばれ、スレッド毎の処理負荷をなるべく均等にすること。
　複数のタスクからなる流れ作業の繰り返しを並列処理する場合、
　負荷分散を考慮しないと、スループットがタスク中の最大の処理時間に
　拘束されてしまいます。

マルチスレッド・プログラミング用語集

「スケーラブルである」とは、
処理データ量やスレッド数の増加による性能劣化がないこと。
コア数増にあわせたスレッド数の増加で、処理時間が短縮
されるプログラムは、スケーラブルです。

「オーバーヘッド」とは、
マルチスレッド・プログラミングでは、スレッド管理コストのこと。
スレッドは、スレッド生成や切り替えなどのスケジュール管理により、
スレッドを利用しない実装よりも処理負荷がかかります。

マルチスレッド・プログラミング用語集

「共有リソース/共有ステート」とは、
　複数のスレッドからアクセスされるリソース(変数/メモリ)を表します。
　読み書き共有となる場合は、内容の整合性を保つため、
　一般的に排他制御による同期が必要です。

「アトミックな処理」とは、
　アトミックとは、これ以上分割できない(割り込めない)ことを表し、
　他スレッドからの割り込みを受けず、一括実行される処理を表します。

マルチスレッド・プログラミング用語集

「スレッドセーフである」とは、
可変なステート(state/状態*1)が、
複数のスレッドからアクセス(読み書き操作)されても、
常に仕様条件を満たしている(*2)ことを表します。

(*1)プログラムでの、変数やデータとオブジェクトの状態など

(*2)increment後は前値より1大きくなる、Aは必ずBより大など

「タスク(task)」とは、
マルチスレッド・プログラミングでは、
並行(並列)に実行できるよう、スレッドセーフかつ、
他処理の結果や副作用に依存しないよう実行する(させる)
独立した処理を表します。

タスクが効率よく並列実行されるための原則
・スレッド数とコア数は同じで、タスク数は左記以上あること。
・タスク内の処理量は、自身のオーバーヘッドよりも大きいこと。

マルチスレッド･プログラミング用語集

「スレッドセーフなプログラム」とは、
複数のスレッドから読み書きされる共有ステートが、
仕様整合性を満たせるよう、適切な同期化が図られて
いるプログラムです。

スレッドセーフか否かのチェックは、一般的結論がありません。
ドキュメントのスレッドセーフ記述有無をチェック(*3)するか、
公開されているステートに対し、複数スレッドから読み書きしても、
リエントラント(再入)を行っても、「仕様が破綻しない」ことを
論理的･具体的にテストするしかありません。

(*3)要利用クラスJavadoc説明「スレッドセーフ」記述有無確認
例:DateForamt　Synchronization 項目
http://developer.android.com/intl/ja/reference/java/text/DateFormat.html

マルチスレッド･プログラミング用語集

「サイズ n の問題/オー記法」(1)とは、
サイズ n の問題とは、計算機科学などで
O(n) (オー･記法)と表記されます。
O(「nの関数」)というのは、計算量がオーダーn関数で
あることを意味します。

例えば、O(n)の場合は、
問題を解くのにnの定数倍の関数で抑えられる
計算(時間)が必要であることを意味します。

注意:オー記法では、計算量の本旨に注目するため、
nが非常に大きな場合に無視できるようになる部分が
省略されているそうです。
(例えば 3n なら 3 が無視できるようになるので n と表記)

マルチスレッド･プログラミング用語集

「サイズ n の問題/オー記法」(2)とは、
「n個の数字を小さい順に並べる」というようなソートの問題の場合、
バブルソート、挿入ソートだったら、計算にO(n^2)時間かかり、
クイックソート、ヒープソートだったら、
O(n log n)時間かかることが判っているそうです。

つまり、オー記法により、
最低でも「O(n log n)時間以下では計算できない」ことや、
バブルソート等では、nが大きくなれば事実上計算不能であることが
示されます。

参考先

ランダウの記号
http://ja.wikipedia.org/wiki/%E3%83%A9%E3%83%B3%E3%83%80%E3%82%A6%E3%81%AE%E8%A8%98%E5%8F%B7

オー･記法(O notation)
http://www.triplefalcon.com/Lexicon/Notation-BigO-1.htm

東京大学理学部情報科学課 計算量理論 授業の内容について
http://www.is.s.u-tokyo.ac.jp/vu/vu_lesson_entry.php?eid=00027&when=3

マルチスレッド･プログラミング用語集

主な参考先

Java 並行処理プログラミング
ソフトバンク クリエイティブ株式会社 ISBN4-7973-3720-6

並行コンピューティング技法
オライリー･ジャパン ISBN978-4-87311-435-4

Java 並行プログラミング機構

実際にマルチスレッド･プログラミングを行う場合、
Java 言語で記述することになると思います。

この項では、細粒度タスク用に利用する
マルチスレッド･プログラミング機構として、
synchronized ブロックや
volataile 修飾子などのプリミティブと、
Java5 からの Atomic変数クラスなどを紹介します。

Java5 のExecutorフレームワークなどは、
粗粒度タスク用のため省略します。

短文化した概要説明となっていますので、
詳しくは、項末の参考資料等で御確認ください。

Java 並行プログラミング機構

synchronized(1)

```
synchronized(ロック･オブジェクト){
    同期処理ブロック
}
```

synchronized は、
同一のロック･オブジェクトを利用した全ての同期処理ブロックで、
ロックを獲得できた、ただ一つの同期処理ブロック(のスレッド)しか
実行されない(同期部は、同時に1つしか実行されない)ことを
保証します。

ロックされた同期ブロックが
実行開始～実行中～実行終了までの間は、
同一のロック･オブジェクトかつ非ロックの他同期ブロックへの
他スレッド突入が抑止(強制待機)されるだけですので、
処理結果の整合性は、プログラマの実装ロジックに委ねられます。

Java 並行プログラミング機構

synchronized(2)

Java言語仕様(JLS)では、
synchronized ブロック内では、実行順序の並び替えや
ロジックの変更などのコンパイラの最適化を抑止して、
指定ブロック内処理の順序の一貫性が守られることを
保証しています。

その他、どのスレッドにおいても、synchronized ブロック内で、
最後に変更された値にアクセスできる(見える)ことが
保証されています。

Java 並行プログラミング機構

synchronized(3)

高コストな同期化を回避するために、
double-checked locking イディオムが考案されましたが、
現行のメモリー･モデルが原因で安全なプログラミング構成
概念ではないことが明らかになっています。

double-checked locking の例

```
public static Singleton getInstance() {
  if (instance == null) {
    synchronized(Singleton.class) {
      if (instance == null)
        instance = new Singleton();
    }
  }
  return instance;
}
```

(スレッド･セーフでありません)

参考先
IBM DeveloperWorks double-checked lockingとSingletonパターン
http://www.ibm.com/developerworks/jp/java/library/j-dcl/

Java 並行プログラミング機構

synchronized(4)

前ページの instance フィールドへの書き込みは1度ですが、読み出しは2度なので、レースコンディション(race condition)(競合状態)により、instance は、コンストラクタで設定した値を返すとは限らないそうです。

スレッド･セーフな getInstance() とする例

```
public static synchronized Singleton getInstance() {
  if (instance == null)
    instance = new Singleton();
  return instance;
}
```

参考先

Effective Java プログラミング言語ガイド
ピアソン･エデュケーション　ISBN4-89471-436-1
第4章 クラスとインターフェース 項目13 不変性を選ぶ

Java 並行プログラミング機構

volatile

volatile 修飾子は、
コンパイラへの最適化抑止(キャッシュ・フラッシュ等)により、
修飾されたフィールド変数を読み出すどのスレッドにおいても、
最後に変更された値にアクセスできる(見える)ことを保証します。
処理順序の一貫性を保証するものではありません。

英語の意味では、揮発性を表し、ハードウェアや他スレッドにより、
非同期に変更されるフィールドである事をコンパイラに宣言します。
コンパイラは、変更が予測不可能であるため最適化を抑止します。

注意:多くのJVMでは、順序の一貫性について
volatile が正しく実装されていないそうです。

Java 並行プログラミング機構

synchronized と volatile の可視性保証

Java 並行処理プログラミング
11-3-2 メモリの同期化 より引用
「synchronized と volataile が提供している可視性保証には
『メモリバリヤ(momory barriers)』と呼ばれる特殊な命令が使われ(16-1-1)、
プロセスのキャッシュをフラッシュまたは無効化し、
ハードウェアの書き込みバッファをフラッシュし、
実行パイプラインを停止します。 メモリバリヤはコンパイラによる
その他の最適化も禁じるので、間接的に実行性能に影響を与えます。
メモリバリヤがあると操作の順序を変える最適化はほとんどできません。」

Effective Java プログラミング言語ガイド
第9章 スレッド 項目48 より引用
「同期は、スレッドがオブジェクトを不整合な状態で調べることを
防ぐだけでなく、連続して実行されているように見える順序づけ
られた状態遷移によって、1つの整合性のある状態から
別の整合性のある状態にオブジェクトが遷移することも保証しています。」

synchronized と volataile には、
可視性の保証があります。

Java 並行プログラミング機構

wait notify notifyAll メソッド

wait notify notifyAll メソッドは、
同一のロック･オブジェクトが指定された
synchronized ブロック内でしか機能しません。

```
ロック･オブジェクト.wait()
ロック･オブジェクト.notify()
ロック･オブジェクト.notifyAll()
synchronized(ロック･オブジェクト){
    同期処理ブロック
}
```

Java 並行プログラミング機構

wait notify notifyAll メソッド

notify notifyAll メソッドの効果は、
synchronized ブロックを抜けないと働きませんし、
そして wait は、すぐに起床するわけではありません。

wait が起こされるまで時間かかるので、
起床時にはロックが奪われているかもしれません。
wait は、while(起床条件不成立) wait(); の
イディオムで使い、起床可能の確認を行う必要があります。

notifyAll() は、安全性が高いが高コストです。

参考先
Effective Java プログラミング言語ガイド
ピアソン・エデュケーション　ISBN4-89471-436-1
第9章 スレッド 項目50 ループの外で決して wait を呼び出さない

Java 並行プログラミング機構

join メソッド

ワーカー・スレッド.join()

join は、join 実行スレッドを
ワーカー・スレッドのスレッドが終了するまで待機させます。

Java 並行プログラミング機構

Thread#yield メソッド

Thread#yield() は、
スレッド･スケジューリングを操作するメソッドです。
yield() やスレッド優先順位の変更を利用しないでください。

スレッドスケジューラは、Java 実行環境に依存します。
どの端末でもスレッドスケジュールが同じとは限らないので、
Thread#yield() メソッドやスレッドの優先順位を指定して、
アプリ側でスレッドスケジュールを操作しようとせず、
OSのスレッド･スケジューリングに任せてください。

参考先

Java 並行処理プログラミング
ソフトバンク クリエイティブ株式会社 ISBN4-7973-3720-6
第10章 生存事故を防ぐ

Effective Java プログラミング言語ガイド
ピアソン･エデュケーション ISBN4-89471-436-1
第9章 スレッド 項目51 スレッドスケジューラに依存しない

Java 並行プログラミング機構

Javaのアトミックな変数

アトミック変数 (正しくはオブジェクト) では、
インクリメント操作も同期ロックなしで一括して行えます。
ですが実装内部では `synchronized` が使われていません。

アトミックな読み書きの基本ロジックは、
変更が割り込みなく成功するまで試行する単純なものです。
CAS（後述）を利用しています。

Java 並行プログラミング機構

Android ソースでの java.util.concurrent.atomic.AtomicLong

```
package java.util.concurrent.atomic;
import sun.misc.Unsafe;

public class AtomicLong extends Number implements java.io.Serializable {
     ～省略～
     //アトミック読み書き中核の unsafe は、Android オリジナルの UnsafeAccess クラス
    // setup to use Unsafe.compareAndSwapLong for updates
    private static final Unsafe unsafe = UnsafeAccess.THE_ONE; // android-changed
     ～省略～
    public final boolean compareAndSet(long expect, long update) {
        return unsafe.compareAndSwapLong(this, valueOffset, expect, update);
    }
     ～省略～
     //Android でもアトミックな読み書きの基本ロジックは、
     //割り込みがなければ変更成功とする、Sun オリジナルと同じ。
    public final long getAndIncrement() {
        while (true) {
            long current = get();
            long next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
    }
     ～省略～
}
```

Android 独自の UnsafeAccess クラスを新規作成して、
アトミックな読み書きに利用しています。

Java 並行プログラミング機構

Android ソースで追加されたクラス
java.util.concurrent.atomic.UnsafeAccess

```
package java.util.concurrent.atomic;

import sun.misc.Unsafe;

final class UnsafeAccess {
    ～省略～
     static final Unsafe THE_ONE = Unsafe.getUnsafe();
    ～省略～
}
```

実は、sun.misc.Unsafe#getUnsafe()を呼んでいるだけ…

自分の仕事を憎むには人生は余りにも短い
sun.misc.UnsafeでFinalなオブジェクトでも書き換える
http://d.hatena.ne.jp/GARAPON/20091116/1258388542
sun.misc.Unsafe クラスは、特定のフ ィールド内容をシリアライズ/デシリアライズすることで任意に変更しているとのこと。

ナンセンス不定期 sun.misc.Unsafe
http://d.hatena.ne.jp/hiuchida/20080703/1215050877
sun.misc.Unsafe クラスの実装は、ネィティブメソッド呼び出しばかり

Effective Java プログラミング言語ガイド
第10章 シリアライズ 項目56 防御的に readObject を書く

*オブジェクトのシリアライズ・バイトストリームに、
攻撃者が偽(任意)のバイト値を書き込んで
デシリアライズさせることで、コンストラクタを
無視したフィールド値を任意に設定できるそうです。*

Java 並行プログラミング機構

高コストな同期化の回避

CAS (compare and swap 比較して入れる)
変更が割り込みなくできたか否かをチェックするだけの
楽観ロックで、読み書きの一貫性を確保するロジックです。
同期によるブロッキングが発生しないので低コストです。

CAS の基本ロジックは、
共有リソースの管理者が、変更要求者からの新旧の期待値と、
管理している実際の値とを比較(compare)することで、
変更要求の整合性可否を検出し(*1)、OKなら管理値を
新期待値と入れ替え(swap)、一貫性をもった変更を保証します。

(*1)旧期待値が以前の状態と一致するなら、最初の変更要求なので、
整合性が一致していると判断できます。
一致しないなら、割り込みがあったことを返します。

Java 並行プログラミング機構

高コストな同期化の回避

アトミック変数でのCASは、
同期と異なり、ロックを奪えなかった競合相手のスレッドが
停止させられることがないような実装(*2)となっているため
変更失敗時のコストも低くなっています。

(*2)Java プリミティブの並行機構だけでは、競合時に
ブロッキングの発生しない整合性制御が行えません。
アトミック変数は、sun.misc.Unsafe を使って、
CASを実現していますが、
Unsafe の中はネィティブコードです。

Java 並行プログラミング機構

参考 **Java7** Fork/Joinフレームワーク

Java7 の fork/join フレームワークは、
ForkJoinPool に渡した ForkJoin 継承のオリジナルタスクを
並行実行させて、マルチプロセッサの能力をフル活用します。
分割統治のアルゴリズムで並列化を行っています。

オリジナルタスクは、大きな作業(処理量/処理)の
再帰的な分割と、待ち行列(デキュー)への追加ならびに、
充分小さく分割された作業の実行を実装します。

ForkJoinPool は、待ち行列(デキュー)から
work-stealing アルゴリズムを利用して、
分割や実行が必要なオリジナルタスクを効率的に取り出し、
利用可能な個数のワーカースレッドでの
オリジナルタスク処理を行わせます。

Java 並行プログラミング機構

参考 **Java7** Fork/Joinフレームワーク

```
//オリジナルタスクの実装イメージ
class オリジナルタスク extends RecursiveAction {
  オリジナルタスク(作業サイズ){
    ～コンストラクタ～
  }
  void compute() {
    if (作業サイズ < しきい値) {
      作業実行();
    } else {
      作業サイズA=作業サイズをA2分割;
      作業サイズB=作業サイズをB2分割;
      invokeAll(new オリジナルタスク(作業サイズA),
                new オリジナルタスク(作業サイズB));
    }
  }
  void 作業実行(){
    ～何らかの作業～
  }
}
```

補足:オリジナルタスクは、値を返すなら RecursiveTask、
返さなければ RecursiveAction を継承させます。
RecursiveTask と RecursiveAction は、
ForkJoinTask のサブクラスです。

Java 並行プログラミング機構

参考 **Java7** Fork/Joinフレームワーク

参考先

Java SE 7 徹底理解
第2回 細粒度の並行処理 - Fork/Join Framework
http://itpro.nikkeibp.co.jp/article/COLUMN/20110527/360769/?ST=develop&mkjb&P=4
要日経BPパスポート会員登録

The Java ™ Tutorials Fork/Join
http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Javaの理論と実践: フォークを活用する、第 1 回
http://www.ibm.com/developerworks/jp/java/library/j-jtp11137.html
fork/join フレームワークや、
ワーク・スティーリング(work stealing)アルゴリズムの説明があります。

Java 並行プログラミング機構

主な参考元

Java 並行処理プログラミング
ソフトバンク クリエイティブ株式会社 ISBN4-7973-3720-6

Effective Java プログラミング言語ガイド
ピアソン・エデュケーション ISBN4-89471-436-1

Javaスレッドメモ (Hishidama's Java thread Memo)
Java マルチスレッド・排他処理
http://www.ne.jp/asahi/hishidama/home/tech/java/thread.html

Android で 並行プログラミング

Android のマルチコア活用ノウハウや、
フレームワークの実験実装の発表を
期待されていたみなさん

大変申し訳ございませんでした。

Android で 並行プログラミング

高い並列性と実行効率を実現するためには、
地道な実装内の依存性確認や依存性排除だけでなく、
独力での並列化手法の実装が必要

消費電力の低減化を目指したいのなら、
CPUが低クロック周波数で定常化するように処理を行わせる
ジョブ･スケジュール対策も必要

同期が高コストなので回避しようにも、
Java プリミティブな並行機構だけでは無理

Android で 並行プログラミング

現状のマルチスレッド･プログラミングが、
うんざりさせらることばかりと思われたでしょうか。

この内容は、現時点(2012/04/21)のものです。

Android のマルチコア化は、始まったばかり
並列タスクが簡易に実装できるフレームワークや、
CPU速度低下指定が、アプリでできるようなAPIは、
明日にでも発表されるかもしれません。

Android で 並行プログラミング

ご清聴、

ありがとうございました。