THE Multicore ASSOCIATION™

# MULTICORE PROGRAMMING PRACTICES

Working Group Chairs: **Rob Oshana, David Stewart & Max Domeika**
President: **Markus Levy**

# Table of Contents

## Chapter 4: Implementation and Low-Level Design



## Chapter 5: Debug

## Chapter 6: Performance



## Chapter 7: Fundamental Definitions

## Appendix A: MPP Architecture Options



## Appendix B: Programming API Options



## Appendix C: Parallel Programming Development Lifecycle



## Appendix D

# FOREWARD: MOTIVATION FOR THIS MULTICORE PROGRAMMING PRACTICES GUIDE

**Markus Levy, Multicore Association President**

The Multicore Association (MCA) の設立は 2005 年 5 月であり、マルチコア技術の萌芽期に遡る。今日、MCA の主たる目標は、広範囲の API 開発と、マルチコアプログラミング手法の実践とサービスに関する蓄積を企業支援によって作り上げることである。

Multicore Programming Practices (MPP) ワーキンググループが 4 年以上前に作られた時、少なくとも次の１０年でも C/C++が支配的であると考えられていた（それは現時点でも同様である）。産業界では、長期にわたるプログラム基盤関連の研究結果を待ち望んでいたが、マルチコア向けプログラミングとの乖離は日に日に拡大していた。そのため、方法論に関して同じ考えを持つ専門家集団によって、標準的な「ベストプラクティス」ガイドをまとめることが強く要望されていた。

Max Domeika (Intel)、Rob Oshana (Freescale)、David Stewart (CriticalBlue)の助言により、MCA は、このユニークな文書を作り出した。この文書は、拡張を使わない C/C++のみを用いたマルチコア向けソフトウェアを書くためのベストプラクティスを与え、逐次コードから並列コードに変換するときに共通して作り出される落とし穴の仕組みについて説明し、バグを減らしデバッグ工数を最小化するための知識をもたらす。

もちろん、この種のガイドを作る時の唯一の問題は、真の意味で「現在進行中」であることである。実際に、この壮大なるプロジェクトの開始時点においてすら、最大の課題は（少なくとも最初のバージョンにおいて）何を含めないか決定していくことであることに気づいた。それゆえ私は、ワーキンググループメンバーに個人的に感謝している一方で、さらに前に進

めるため、皆さんを、MCA に入り、この MPP ガイドの進化を助けてくれるよう招待したいと思っている。

MPP ガイドの主たる貢献者は以下の通りである。

Hyunki Baik (Samsung)
François Bodin (CAPS entreprise)
Ross Dickson (Virtutech/Wind River)
Max Domeika (Intel)
Scott A. Hissam (Carnegie Mellon University)
Skip Hovsmith CriticalBlue)
James Ivers (Carnegie Mellon University)
Ian Lintault (nCore Design)
Stephen Olsen (Mentor Graphics)
David Stewart (CriticalBlue)

# CHAPTER 1: INTRODUCTION & BUSINESS OVERVIEW

## 1.1 Introduction

マルチコアプロセッサを利用できるようになってから何年も経過した。しかしながら、様々な製品、複数の市場で使われるようになったのはごく最近のことである。この「マルチコア時代」の到来は、シングルコアプロセッサが電力・性能限界に至ったことが原因である。マルチコア時代において、性能向上に対する責任の多くはソフトウェア開発者に移る。ソフトウェア開発者は、どのように処理をコアに分散させるか、指示する必要がある。将来的には、一つのプロセッサ上に集積されるコアの数は増えると予想され、ソフトウェア開発者にかける負担も大きくなるだろう。コンピュータ業界における、これまでの多くの新しい時代と同様、開発者が最小の努力で最大の性能が得られるよう、多くのソフトウェア開発支援ツールと技術が導入されてきている。開発者を妨げる潜在的な課題の一つに、マルチコアプロセッサ向け開発の本質と既存ソフトウェアの慣性との対立がある。

マルチコアプロセッサ時代より前、プロセッサの内部クロック周波数向上やマイクロアーキテクチャ改善を施したプロセッサのアップグレードを通して、開発者は性能を向上させることができた。これらのアップグレードは、最低限のソフトウェア変更のみを要求した。マルチコア時代において性能を向上させるためには、開発者は、現状の逐次的なアプリケーションから並列的なものへと変更するような、ソフトウェアの大幅変更に力を注ぐ必要がある。この変更は些細な事ではない。プログラムの分析・設計・実装・デバッグ・性能チューニングといった、これまでのソフトウェア開発における複数の開発段階にまたがる新たな課題を持ち出すことになる。

全く対照的に、既存ソフトウェアの慣性は否定しがたいものになっている。厳しい納期を満たすために、アプリケーションに対する大きな変更を施す自由を開発プロジェクトが持つことは稀で、逆にプログラム変更を最小限にすることでリスク増を防いでいる。このことは、マルチコアプロセッサへ移行することを考えている開発者が、新しい並列プログラミング言語を採用する、あるいは広範囲な並行実行を可能にするようアプリケーションを再構築する、といった余裕を持たないことを意味している。その代わり、多くのコンピュータ業界の開発プロジェクトはマルチコアプロセッサを活用できる段階的なアプローチを採用している。このアプローチでは、既存のプログラミングツールと技術を利用し、既存のソフトウェアに並行性を導入し、デバッグするための体系的な手法を適用する。

MCA MPP ガイドはマルチコア開発にこの段階的アプローチを使用するためのベストプラクティスの詳細である。次節以降では MPP ガイドの目的、対象とする読者、MPP ガイドの適用方法、カバーしていない範囲についての詳細を述べる。

## 1.2 Goal of MPP

MPP ガイドでは、既存技術と既存ソフトウェアを使用してマルチコアプロセッサ向けソフトウェア開発を行うためのベストプラクティスのいくつかについて詳細を述べる。この文書を書くにあたり、MPP ワーキンググループはこのドキュメントの目標が何で、どの程度の分量にすべきか妥協と決定を行った。

高レベル決定のいくつかは以下の考えに基づく：

- APIの選択 ― PthreadsとMCAPI（Multicore Communications API）については、いくつかの理由によって選んだ。これら2種類のAPIにより、マルチコアプログラミングの主要な2種類の分類、すなわちSMPとメッセージパッシング、がカバーされること、PthreadsはSMPに使用されているとても一般的なAPIであること、MCAPIは最近導入されたAPIであるが、ホモジニアス、ヘテロジニアス向け組込みアプリケーションに対する、これまでにないメッセージパッシングベースのマルチコアAPIであること、が理由である。
- アーキテクチャ分類 ― アーキテクチャの3分類、共有メモリ型ホモジニアスマルチコア、共有・非共有メモリ混在型ヘテロジニアスマルチコア、非共有メモリ型ホモジニアスマルチコア、はこの順で優先度が高い。これは、組込みアプリケーションにおけるグループ評価を反映している。正直なところ、第2のカテゴリは第1、第3カテゴリの上位集合である。3つのカテゴリに分けて並べる動機は、第2のカテゴリが、第1及び第3カテゴリに関連した技術、すなわちSMPの技術とプロセスレベル並列性、の広範囲かつ共通的な利用のためである。次の段落でこの目標を分析し、より詳細な情報を提供する。

## 1.3 Detailed Description

このベストプラクティスは、マルチコアプロセッサ向け開発タスクを完遂するためによく知られた方法の集合体である。その意図は開発技術の共有である。それらの開発技術はマルチコアプロセッサ向けに効果があることが知られている。これらの技術の適用により、タイム・ツー・マーケットの短縮とより効果的な開発サイクルを通して、開発コストの削減という結果をもたらす。

MPP で議論されるソフトウェア開発フェーズ、および各フェーズの概要は以下の通りである（訳者注：はじめの 4 項目は章立てに対応している）：

- プログラム解析と高レベル設計とは、並行性を追加する箇所を決定するためのアプリケーション調査と、並行性を持つようなアプリケーションに修正するための戦略である。
- 実装と低レベル設計とは、デザインパターン、アルゴリズム、データ構造の選択と、並行性のソフトウェアコーディングである。

- デバッグには、並行性に起因する潜在的問題を最小限に抑えるような並行性の実装をすること、それらの問題に関してアプリケーションを簡単に調査できるようにすること、そしてそれらの問題を見つけるための技術についても含む。
- 性能は、アプリケーションの応答時間やスループットの改善に関係する。これは、通信、同期、ロック、負荷分散、及びデータ局所性に関わるボトルネックの影響を見つけ、対処することにより実現する。
- 既存技術はプログラミングモデルとマルチコアアーキテクチャを含み、このガイドで詳述するが、今日広く使われている少数に限定している。
- レガシーソフトウェアとして知られる既存ソフトウェアとは、ソフトウェアコードによって表現され、現在利用されているアプリケーションのことである。既存ソフトウェアを使用している顧客は、マルチコアプロセッサを有効に活かすためにアプリケーション全体を再実装するのではなく、新商品開発のために現在の実装を発展させることを選択している。

## 1.4 Business Impact

MPP ガイドの読者は、ガイドで説明する詳細な段階を適用することによって開発プロジェクトのコストを大幅に削減する可能性がある。実装の分析・設計の効率化、実装におけるバグ数の減少、性能要件に見合う実装可能性の増加によって、開発者はコスト削減できる。この削減はタイム・ツー・マーケット短縮と開発コスト低減へと導く。

## 1.5 Target Audience

MPP ガイドは特に、マルチコアプロセッサを使い、既存のマルチコア技術を使用する開発プロジェクトを計画、実施する企業の技術者及び技術マネージャのために書かれている。

以下では想定読者に対するこのガイドの恩恵について具体的にまとめる：

- 逐次プログラミングの経験があるソフトウェア開発者は恩恵を受けるだろう。このガイドにおいては、マルチコアプロセッサ向けソフトウェア開発を行う際に必要な新しい概念が説明されている。
- ハードウェアおよびソフトウェア技術者チームのマネージャはこのガイドを読むことで恩恵を受けるだろう。マルチコアプロセッサ向け開発およびソフトウェア開発者が直面する学習の難しさについて知ることができる。

- 複雑なプロジェクトを計画し実行するプロジェクトマネージャはこのガイドを読むことにより恩恵を受けるだろう。マルチコアプロセッサを利用するプロジェクトについて理解し、適切な計画を立てることが可能になる。
- 機能検証、性能検証のテストを開発するテスト技術者はこのガイドを読むことにより恩恵を受けるだろう。マルチコア関連プロジェクトのためのより適切でより効果的なテストを作成することが可能になる。

## 1.6 Applying MPP

MPP ガイドはソフトウェア開発段階にもとづいた章立てになっている。各章は、簡単な紹介に続いて、開発プロセスの説明、あるいは技術的な詳細を含む話題ごとの説明、もしくは両方を含んでいる。MPP ガイドは、それぞれの話題に関する大雑把な説明以上のものを提供し、平均的な開発者に向け、簡潔だが十分に詳しい説明を提供することに努めている。

MPP ガイドにおけるコード例は実際の実装に基づいており、与えられた開発ツールで再現するための解説を加えている。Figure 1 は C コードとコンパイルのための解説を示すサンプルコードのリストである。上の枠はサンプルの C コードである。ソースコード内のコメント‘/* File:’に続くテキストは再現するための手順で参照されるファイル名であり、この場合 example.c である。‘To reproduce:’に続くテキストは再現するためにコマンドライン上で入力するコマンドである。このガイドのコード例は同じフォーマットに従う。

```
/* File: example.c */
#include <stdio.h>
#include <pthread.h>

void *hello (void * arg) {
      printf("Hello Thread\n");
}

int main() {
  pthread_t tid;
  pthread_create(&tid, NULL, hello, NULL);
pthread_join(tid, NULL);
return 0;
}
```

To reproduce:

```
      cc -o example
example.c -lpthread
```

**Figure 1. Code sample and steps to reproduce.**

## 1.7 Areas Outside the Scope of MPP

MPP ガイドは、（あまりにも多くの異なる技術や方法がある）並列プログラミングのためのすべての可能な仕組みを説明しているわけではない。ガイドは、従来から一般的に使用されている手法の範囲に限定している。このガイドで詳述する多くの手法は、一般に他のプログラミングモデルにも適用できるが、このガイドは他のプログラミングモデルへの変換に必要な詳細については説明しない。第 7 章には多くの技術及び方法についての説明があるが、議論は一般的な話にとどめ、その前の章で記載されているベストプラクティスに特化したものではない。

特に、以下は MPP ガイドにおけるベストプラクティスの範囲外である：

- C/C++以外の言語、ISO標準ではないC/C++拡張、独自のC/C++拡張
- コーディングスタイルガイドライン
- 第2章で述べるアーキテクチャ、プログラミングモデル、C/C++互換並列化ライブラリ以外のもの

# CHAPTER 2: OVERVIEW OF AVAILABLE TECHNOLOGY

## 2.1 Introduction

マルチコアプログラミングツールおよびモデルには、ソフトウェア開発ツール、プログラミング言語、マルチコアプログラミング API、ハードウェアアーキテクチャのような項目を含んでいる。このガイドに記載されているプログラミング手法は特定のツールやモデルを例として用いているが、われわれの目的はこれらの手法を一般的に適用できるようにすることにある。この章では、後の章で説明するプログラミング手法において想定される、プログラミング言語、マルチコアプログラミング API、ハードウェアアーキテクチャについて述べる。

## 2.2 Programming Languages

標準 C および C++は組込みソフトウェア開発に使用される主要な言語であるので、MPP ガイドでは実装言語として標準 C および C++を採用している。「ISO/ IEC9899:1999、プログラミング言語 C」で仕様化されている C 言語（いわゆる C99）、および「ISO/ IEC14882、標準 C++言語」で仕様化されている C++言語が使用される。どちらの言語においても、我々は商用または研究のいずれでも非標準の拡張機能を使用していない。

## 2.3 Implementing Parallelism: Programming Models/APIs

マルチコアプログラミングで利用される並列性にはいくつかの種類、タスクレベル並列性（Task Level Parallelism (TLP)）、データレベル並列性（Data Level Parallelism (DLP)）、命令レベル並列性（Instruction Level Parallelism (ILP)）、がある。MPP ガイドでは、TLP と DLP に焦点を当てるが、三種類の並列性はすべて設計と実装の決定に重要な役割を果たす。

MPP ガイドではマルチコアプログラミング API の例として、Pthreads[i] とマルチコア通信 API（Multicore Communications API (MCAPI[ii])）を使用する。これはマルチコアソフトウェア開発における二つの種類、共有メモリプログラミングとメッセージパッシングプログラミング、の両方をカバーすることを可能にする。Pthreads と MCAPI は、2 種類のソフトウェア開発手法においてそれぞれ一般的であるため、MPP ガイドではこれら 2 つの API を選択した。

## 2.4 Multicore Architectures

MPP ガイドでは、3 種類のマルチコアアーキテクチャ、共有メモリ型ホモジニアスマルチコア、共有・非共有メモリ混在型ヘテロジニアスマルチコア、非共有メモリ型ホモジニアスマルチコア[iii]、を対象としている。これら 3 種類は組込みプロジェクトで用いられる主要なア

ーキテクチャであり、各々十分に違いがあり、適用事例が本質的に異なっている。下回りの通信機構はこのガイドの範囲外であり、ほとんどの場合はソフトウェア開発者とは無関係である。

共有メモリ型ホモジニアスマルチコアプロセッサは、同じ ISA を実装し、同じ主記憶メモリへのアクセスを持つ複数(2 から n)のプロセッサコアで構成される。

共有・非共有メモリ混在型ヘテロジニアスマルチコアプロセッサは、異なる ISA を実装し、プロセッサコア間で共有される主記憶と各コアのローカルメモリとの両方にアクセスできる複数のプロセッサコアで構成される。このタイプのアーキテクチャでは、ローカルメモリはシステムの一部からのみアクセス可能な物理記憶領域となる。例として、Sony/Toshiba/IBM による CellBE プロセッサ内の各 SPE に関連するメモリがあげられる。一般にメモリは、プロセッサコア、プロセッサ、SoC あるいはボードのいずれに対してもローカルである可能性があるため、メモリがシステムのどの部分に対してローカルであるかを記述することが必要である。

非共有メモリ型ホモジニアスマルチコアプロセッサは、同じ ISA を実装し、非共有でローカルな主記憶メモリへのアクセスを持っている複数のプロセッサコアで構成される。

## 2.5 Programming Tools

このガイドで用いられるソフトウェアツールおよび使用法の詳細は、以下のように要約される：

- GNU gcc version 4.X - すべての C 言語の例に使われるコンパイラ
- GNU g++ version 4.X - すべての C++言語の例に使われるコンパイラ

# CHAPTER 3: ANALYSIS AND HIGH LEVEL DESIGN

## 3.1 Introduction

この章で議論するプログラム解析は、並列動作を活用できる条件やその動作を制限する依存関係について明らかにすることを目的としている。使用するベンチマークやその処理量の現実味は解析結果に大きく影響する。アルゴリズムやプラットフォームアーキテクチャの選択、適切な並列化デザインパターンの決定などの高レベルの設計判断が行われ、性能や電力、コードサイズ、スケーラビリティなどの重要なアプリケーション指標を満足するようにマルチコア実装を最適化する。

## 3.2 Analysis

マルチコアを効率的に活用するためには、プログラムを並列化する必要がある。しかし、最適化を導入する前にいくつか踏んでおくステップがある。この節では、そのようなステップのうちの二つ、逐次性能向上[iv]とプログラム理解、について説明する。

## 3.3 Improving Serial Performance

マルチコアを活用するためのプログラム再設計の最初のステップは、プログラムの逐次実装の最適化である。なぜなら、逐次性能チューニングは通常は容易であり、時間がかからず、バグを作りこむことも少ないためである。逐次性能改善は、現状と性能目標との差を縮め、そのことは並列化の必要を減らすことを意味することになろう。これらの最適化により、逐次と並列の課題が混じることなく、後に並列化に焦点を当てることができる。

しかし、逐次最適化が最終目的ではないことを忘れないことは重要である。最適化では、並列化およびそれがもたらす性能向上を容易に得るためのプログラム変更のみを、慎重に適用したい。特に、並列化を阻害したり、制限したりする逐次最適化は避けるべきである。例えば、不必要なデータ依存を作りこんだり、（キャッシュ容量などの）シングルコアハードウェアアーキテクチャの細かい機能を利用したりすることは避ける必要がある。

開発者はプログラムの逐次性能を向上させるために使われる技術について熟知している。実際、論文や本、ツール、蓄積された知識といった公開資料は大量に存在する。しかし、そのような改良を場当たり的に適用することは良いとは言えない。

プログラマは、プログラムのクリティカルでない部分の速度について心配したり考えたりして、多くの時間を浪費している。そして効率性に関するこれらの試みは、デバッグや保守のことを考えると、実際にはとても悪い影響がある。小さな効率化については忘れるべきである。ざっと 97%の時間を占めている先走った最適化は諸悪の根源となっている[v]。

その代わり、方法論と反復手法が効率的な逐次性能チューニングの鍵となる(Figure 2)。方針決定のために測定と慎重な分析を行い、一度に一箇所だけ変更し、そして変更が有益であったことを確認するため注意深く再測定を行うべきである。

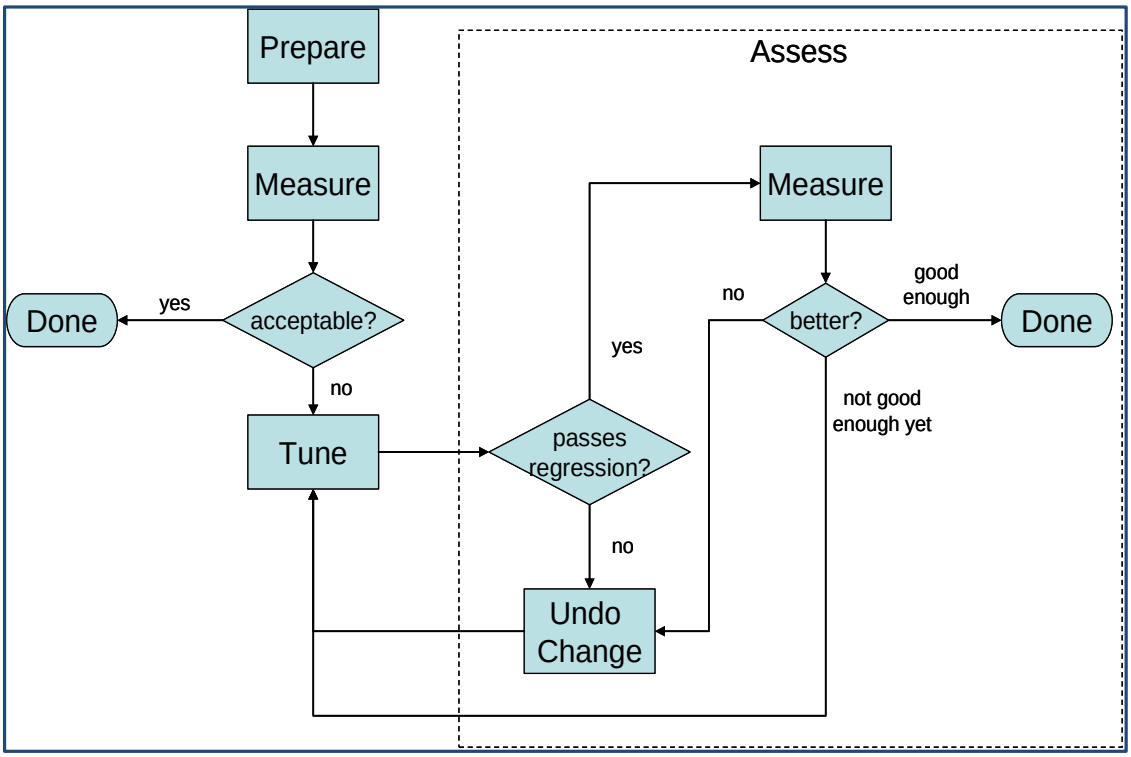



**Figure 2. Serial performance tuning approach. Four key steps: prepare, measure, tune, assess.**

以下の章では、この手法の各段階について簡潔に説明するが、それぞれの段階には、この文書に書かれていない、より深い内容がある。性能解析や性能チューニングに対しては有用な取り組みが数多くあり、そのほとんどがマルチコアの準備として適用可能である（しかし、並列化のためのプログラム変更の効果は、変更の適用順序に大きく影響される）。

### 3.3.1 Prepare

まず回帰テストプログラムとプログラム性能向上目標に対応したベンチマークを集める必要がある。プログラムを変更する前に、その目的について考え、効率的かつ確実に目的を達成するのに役立つ材料を集めるべきである。

- 目的は何か？プログラムの性能向上は複雑で、時間がかかり、エラーを起こしやすい仕事になり得る。「十分な結果」を得られたら、作業は終わりにすべきである。
- 最も重要な実行シナリオは何か？　どのような改善も、プログラム実行結果の観測データに基づいて施されることにはなるが、最も重要なシナリオは何だろうか？　それは、

安定状態下での実行、高負荷状況での実行、エラー回復シナリオ、またはそのようなシナリオの組合せかもしれない。

- どのような性能測定基準があなたのプログラムにとって意味があるのだろうか？　もしかすると、それは単位作業の処理時間かもしれないし、単位時間に処理できる作業量かもしれない。測定すべきことを知ることで、目標の定量化や、目標達成を判断することができる。

性能データを集めるベンチマークセット（特定のデータと演算を持つプログラムの利用）を作成する必要がある。すべての最適化はベンチマークに関連するため、性能向上を最も必要とするシナリオを慎重に選ぶべきである。また、様々な処理量のデータを含めることにより、性能スケールを見ることができる（線形と非線形を区別するためには、少なくとも 3 つのデータが必要となる）[vi]。

性能最適化はときにメモリ消費、コード保守性、移植性といったその他の関心事とのトレードオフとなる。どれにどのような妥協をしたいのかについて明確にしておく必要がある。

逐次性能チューニングは、性能の計測と機能の正しさを保証するため、プログラムの繰り返し実行を含む反復プロセスとなる。そのため、始める前に適切な回帰テスト群があることを確認する必要がある（始める前にプログラムがすべてのテストにパスすることは言うまでもない）。

### 3.3.2 Measure

準備の後、次にやるべきことは、ベンチマークを使って基準となるプログラム性能を調べることである。プロファイラは性能測定のツールとして一般的に使われている。プロファイラとはプログラムの実行を観測し、性能測定結果を収集するためのツールである。プロファイラにはいくつかの種類がある：

- サンプリング：サンプリングプロファイラは定期的時間間隔で計測する[vii]。
- 計測化 (Instrumentaion)：計測化プログラムを用いたプロファイラは一般に、計測のための命令を追加し、プログラムと一緒にコンパイル、リンクする。
- エミュレーション、シミュレーション：（インタープリタ等により）実行環境をエミュレートまたはシミュレートし、その上でプログラムを動作させる。

計測方法によって性能データの正確さはいくつかの点において異なる。それらのうち二つは完全性と「真の性能」との類似性である。定期的な時間間隔サンプリングだけでは、プログラム動作について不完全な情報しか得られない。サンプリング以外の方法であれば、計測精度の範囲内で、完全なデータを生成できる。しかしながら、いずれのプロファイリング手法も、計測無しの実行との類似性に関して何らかの影響が出る。オーバーヘッドが小さい方法

は実際の実行とのより高い類似性がある。計測化プログラムやインタープリタなどオーバーヘッドの大きい方法は、プログラムの実行時間に関する性質を大きく変えてしまう可能性がある。

出力はツールごとに精度やフォーマットが異なる。プロファイラは、典型的には関数に関するデータを、関数の呼び出し時と戻り時に単純な計測を行うことにより、計測する。プロファイラによっては、より細かく測定できたり、ユーザが計測点を（例えば調べたいループを網羅するために）追加したりできる。

プロファイラ出力は、計測情報のフィルタリングやソートが行われ、単純なレポートから高度な GUI まで様々なものがある。以下に一般的な出力形式を二つ示す：

- フラットプロファイルは一般に、関数の処理時間（割合や時間単位）や関数が呼ばれた回数といった、関数の統計情報に関する表形式のレポートである。
- コールグラフは、やはり関数単位に、関数呼び出し連鎖に関連して実行時間を示す。与えられた関数に対し、その関数が呼んだ関数、その関数が呼ばれた回数および実行時間を示す。

通常は両方の出力形式を使いたくなるだろう。フラットプロファイルは、実行時間のうち大きい割合を消費している場所（例えば 60%の時間を消費している関数）を手早く見つけることに適している。コールグラフは、ある関数がどのタイミングで使用されているかを調べるのに適している。

プロファイラは、利便性やコスト、ターゲットプラットフォーム、経験など多くの判断基準に基づいて選択される。性能に関するより完全な状況をつかむため、対象プログラムの外部（例えばシステムコール）での実行時間を計測するためのプロファイラを少なくとも一つは使用すべきである。組込みシステム開発においては、ターゲットハードウェアが使えるようになる前や、ターゲットデバイス内の計測用記憶容量不足を補うために、エミュレーションやシミュレーションを利用したプロファイラを使う必要があるかもしれない。

### 3.3.3 Tune

性能チューニングを行う上で「一度に一箇所しか変更しない」ことは重要な経験則である。チューニングの毎回の繰り返しにおいては、プロファイリングを通して収集した計測結果を用い、妥当な労力で最大の性能向上が得られる場所を探すことになる[viii]。忘れてならないことは、この後、プログラムを並列化するために相当な変更を施すことである。そのため、逐次プログラムの性能改善時には、簡単にできること、もしくは大きく性能改善できることのみを施すことが、一般に最善である。例えば、実行時間の大きな割合を消費する部分を、対数的ではなく線形的に時間変化するようなアルゴリズムに変更することは、注目すべき変更

のよい例だろう。一般に、データ使用量を最小化する変更よりも計算を削減する変更の方が好まれる。この段階では、正確に機能させるために必要不可欠な計算のみをするようにプログラムを削減することに焦点を当てるべきである。

まずは、ホットスポットの計測から開始する。ホットスポットとは、プログラムの中でコードサイズのわりに実行時間の割合が特に多い部分であり、フラットプロファイルにより見つけることができる。ホットスポットから始めるのは単純な理由で、頻繁に実行されるコードの改善は、まれに実行されるコードに似たような改善を施した場合と比べると、全体として効率よく改善されるためである。例えば、全体の 50%を占めるコードを 10%改善した場合、全体として 5%改善されるが、全体の 5%を占めるコードを 10%改善した場合、全体として 0.5%しか改善されない。収穫逓減（diminishing return）に注意すべきである。

検討するホットスポット選択の後、その場所がなぜそれほどまでに時間を消費するのか、そしてそれは妥当なのか考えるべきである。

- 計算：プログラムを実行してみることは私たちがまず最初に思いつくことである。
- 適切なアルゴリズムが使用されているか？：様々な負荷に対するプロファイリングデータを調べ、予想した通りにスケールしているか確かめる。
- すべての計算が必要なのか。計算をループ外に移すことができるかもしれない、計算を減らすためにルックアップテーブルを使用できるかもしれない。
- 正しい型やデータ構造を使っているか。一般に、メモリ上での大きなデータ転送には時間がかかる（例えばdouble 型を使うより float 型を使った方が多くの場合速くなる）。利用プラットフォームにおいて異なる言語構造が与える影響を考えてみる（例えば、例外処理をサポートするために必要なオーバーヘッドなど）。
- プログラム分割は適切か。callやreturnにかかるオーバーヘッドよりも十分大きな実行時間にならない関数があれば、インライン展開やリファクタリングを行うべきである。
- メモリの永続的な利用について考えるべきである。構造体（またはクラス）の再利用や再初期化は、新規作成よりも効率的である。
- 待ち：待ちを行う必要がある処理もある。それは特に逐次プログラムで問題となる。ネットワークやファイルシステムアクセスのような I/O 処理には時間がかかる。ユーザとのやりとりも同様である。キャッシュを使用した方法によって多少改善するが、並行性を利用することが典型的な解決策である。ここでの並行性導入は避けるべきだが、依存関係には注意すべきである。これは、後に並列化戦略として考慮すべきことである。
- 大きなデータを使う処理は、メモリに対し頻繁にデータの入出力を行う原因となる。不必要なデータの削除により改善されるかもしれない。メモリプロファイラはメモリの使用状況を理解し、改善するための優れたツールである。

- 逐次性能が改善されるとしても、局所性を考慮したデータ再配置（同時に使用されるデータをメモリ上で近い位置に保ち、データをメモリから何度も転送する代わりに一度で行うこと）は避けた方がよい。この種の最適化は、キャッシュ幅やポリシのような要因を考慮し、並列化の一部として進めた方が良い性能が達成できる。

それぞれのホットスポットを調べると同時に、ほかのいくつかの情報も確認すべきである。コードからはわからないことでも、プロファイラが生成するコールグラフから多くの情報を得ることができる。関数呼び出しを見ることで、その関数が何をしているのか、またどのように使用されているのかがわかる。特に、コールグラフを戻っていくことで（関数の呼び出し側をたどっていくことで）コールグラフの葉ノードではないホットスポットを見つけることができるかもしれない。

### 3.3.4 Assess

プログラムを変更するたびに、一度止まって効果を評価すべきである。「良い変更」とは、バグを作りこまず、プログラムへの追加変更が、直接的または間接的に、準備段階(3.3.1)で示した性能評価を向上させることである。回帰ソフトウェアパッケージの再実行や、ベンチマークを使用したプログラムの再計測により、これらの基準について評価すべきである。

回帰テストに失敗したり、性能が変更前よりも悪くなったりしたならば、プログラムや変更について勘違いがある。失敗の原因を理解するため、変更前後のプログラムをもう一度調べる必要がある。もし修復できなければ、チューニング段階(3.3.3)まで変更を戻すべきである。次のホットスポットに取り掛かる前に、そのホットスポットをもう一度調べ、別の変更を試してみるべきである。

回帰に成功し、性能が改善されたら、その変更が他の懸念事項にも問題がないことを確認する必要がある。例えば、プログラムがもともと持っていた移植性を維持することが大切かもしれない。こういった基準について自動的に検証することは難しい場合もあるが、心のうちに留めておくべきことである。

## 3.4 Understand the Application

逐次処理から並行処理へのアプリケーション変換手法に依らず、解こうとしている問題の十分な理解は避けて通れない。このような理解は、文書、アプリケーション開発者との議論、プロファイルデータを含む様々な情報から得ることができる。

次節以降で説明するように、アプリケーションの振る舞いに関するいくつかの側面は、計算やデータの依存関係のように、並列化に特に関係する。優れたアプリケーション解析は、も

との問題を解くために本質的な依存関係か、人為的、おそらく逐次最適化の一環として導入された依存関係か、判別することに役立つ。

### 3.4.1 Setting Speed-up Expectations

並列化のためにアプリケーションを分割および再設計することには一般的に２つの理由がある：大規模問題を解くことと高速化である。大規模な問題を解くためには、複数コアに作業を振り分けるだけではなく、（時に）シングルプロセッサには問題が大きすぎるほどの大規模データを複数プロセッサに分散させる[ix]。高速化はここでの議論の主要な原動力である（マルチコアプラットフォームに移植されたレガシーアプリケーションが、すでにデータやメモリ要件に対して適切な大きさになっているという仮定に基づく）。

コアに効率よく負荷分散して実行時間を最小化することにより高速化は達成されるが、限界がある。例えば、効率よく並列化できないコード部分がある。Amdahl の法則（6.2 節を参照）は、アプリケーションの一部の性能改善によって予想される最大の効果を表す。ある部分の逐次実行時間が総時間の 80%を占め、その部分を 8 コアに分散する場合、高々3.3 倍高速化できる。同じ箇所が総時間の 20%しか占めていない場合、同じ並列化でも高々1.2 倍しか高速化されない。これらが予想される最大の性能向上であるが、様々なオーバーヘッド要因（協調動作のためなど）がより一層効果を下げる。

アプリケーションを並列化する労力は、アプリケーションの客観的な分析と変更に必要な労力からなるが、その労力は、結果として得られる現実的な期待値によって正当化されなければならない。

### 3.4.2 Task or Data Parallel Decomposition

アプリケーションのプロファイルから、計算がより集中している箇所（例えば、逐次アプリケーションの計測を用いて生成したフラットプロファイルから特定したホットスポット）を見つけるべきである。これらの箇所は、計算の性質により、2 種類の一般的な並列分割形式、タスク並列分割またはデータ並列分割、のどちらかに分類される（注：タスク並列分割は時々機能並列分割と呼ばれる）。

タスク並列分割は、問題の解決法が並行実行可能な一連の逐次処理として考えられ、それぞれが完全または相対的に独立していて、タスクの実行順序に関係なく結果が同じとなる場合の方法である。完全な独立とは、タスク同士がデータを共有せず、また処理間同期が必要ない状況を意味する（極めて珍しい）。相対的な独立とは、比較的よく起こる状況で、タスク同士がデータを共有し、特定の時間に同期を行う必要がある状況を意味する。タスク並列分

割の一例には画像処理とキャプチャーからなるアプリケーションがあり、カメラからビットマップを受信して 3 つの操作を実行する：

- 画像内の色の数を数える。
- ビットマップからJPEGに変換する。
- ディスクに画像を保存する。

逐次アプリケーションではこれらの処理が順次実行される（Figure 3）。タスク並列化されたアプリケーションでは、依存関係がないため 2 つのタスクが並列実行可能であるが、画像を保存する 3 つ目のタスクはビットマップを JPEG に変換した後にのみ実行できるため、エンコード処理のタスクと画像保存のタスク間で同期が必要である。アプリケーションが変換した JPEG ではなく、もとのビットマップのみを保存する必要がある場合には、3 つのタスク間の依存関係が完全になくなり、3 つすべてのタスクの並列実行が可能である。

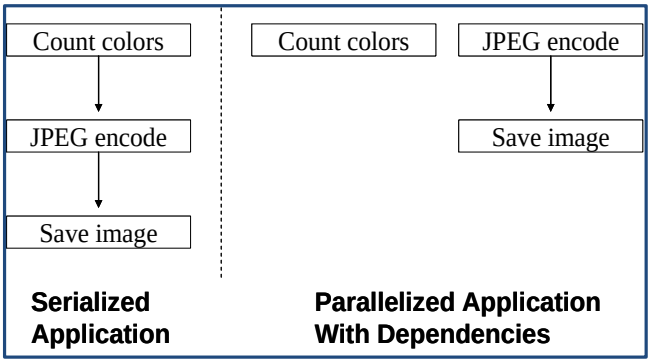



**Figure 3. Task parallel decomposition example.**

データ並列分割は、問題の解決法が主要なデータ構造（配列等）を独立した小さなデータブロックに細分化・分割して、独立に並行して実行できる場合の方法である。このような場合の例として、行列のスカラー倍演算がある（Figure 4）。逐次アプリケーションでは、多くの場合このような演算はループ構造を利用して逐次実行される。この例では、行列内のほかの要素の演算とは完全に独立してそれぞれの要素を演算することができる。さらに、演算を任意の順番で実行することができ、どのような場合でも演算の結果は同じになる。逐次的なループ構造によって定められる順序関係は、単にループ構造の意味によって与えられただけの人為的なもので、同じ行列演算を計算する並列形式はこのような制約を受けない。このように、行列の各要素は並列化によって分割された形式となり得る。行列が 2x2 である場合、各要素に対応するスカラー倍演算を実行する 4 つの関数が存在する。

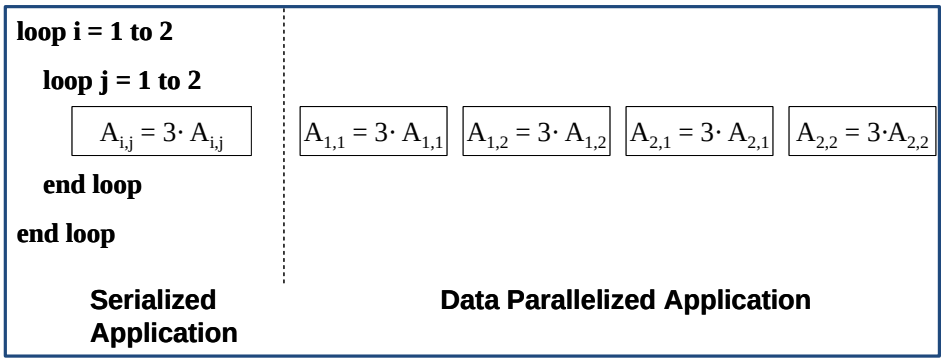



**Figure 4. Data parallel decomposition example.**

タスクまたはデータ並列分割では、一方が他方に影響を与えることが多い。つまりタスク並列分割とデータ並列分割は互いに排他的ではない。ほとんどのアプリケーションの問題は両方の形式に適用可能であり、どちらの形式が適しているか知るためには問題を完全に理解することが必要である。画像処理とキャプチャーからなるアプリケーションのような問題はタスク並列分割がより適しているが、その他（スカラー倍等）の問題にはデータ並列分割がより適しているものもある。いずれの並列分割にしてもデータ構造は重要であり、タスク並列分割では並列分割したタスクがそれぞれの計算を実行できるように、データ並列分割では分割によって生成される複数の独立ブロックを処理する複数のタスクを生成できるように、データ構造を設計しなければならない。

### 3.4.3 Dependencies, Ordering, and Granularity

依存関係は、逐次アプリケーション内にある処理間の順序関係か、処理間にあるデータ依存関係（例えばメモリ共有）のいずれかの形式である。プロファイラのなかには、アプリケーション内にある処理やタスクの実行順序関係やその頻度を調査するのに役立つコールグラフを生成できるものもある。アプリケーションの逐次的記述によって処理やタスクの実行順序がどうなるか見通しが立つかもしれないが、シングルプロセッサ上で逐次アプリケーションを最適化した際に妥協があったかも知れず、また、ほかの（優先する）目標を達成するために人為的に順序関係を入れる場合もある。アプリケーションの逐次的記述から離れ、並列化の可能性を洞察することは、処理間の偽の順序（必要のない順序）を見出し、並列化できる箇所を見つけるための助けとなるだろう。上記の画像処理とキャプチャーの例では、ビットマップ内の色の数を数えることは必ずしも最初に実行する必要はない。

コード上の異なる部分（おそらく大きい部分）が共有するデータ依存関係もコールグラフで識別できるかもしれないが、このようなデータの依存関係に対し、その真偽を確かめるソースコード検査が必要となる可能性が高い。静的解析ツールやメモリプロファイリングツールはコードの広範囲に対してデータの依存関係を識別する（例えば、膨大な行数のソースコードや複数のソースコードファイルから大域変数や共有メモリ領域などへのアクセスを探索す

る）目的で使用することができるかもしれない。このようなツールは言語に依存し、もしかするとプラットフォームにも依存する可能性がある。

計算が集中するループによるホットスポットでは処理内に依存関係を作り出しているかもしれない。ループ処理が並列実行できるように設計されている場合、高速化が可能になるため、このようなホットスポットは並列化の候補となる。しかし、これを実行するためには、各ループ処理を独立させるように、つまり、各繰り返し処理が任意の順序で安全に実行でき、繰り返し処理間に依存関係が発生しないように、ループを解析、再設計しなければならない。一方で、逐次アプリケーションのループのなかには繰り返し処理間に依存関係がなくてはならないものもある。例えば、蓄積計算のような連想操作の実行では依存関係の消去が難しい。

高速化方法を見つけ、すべてのコアの効率的利用を達成するためには、作業（タスク並列分割またはデータ並列分割）実行に必要とされるタスク分割、順序付け、設計そして実装において、ターゲットプラットフォームを意識しなければならない。このことは、これらのタスクのために生成された処理が、コア使用率を維持する十分な大きさであり、かつ複数コア間で良好に負荷が均衡することを保証することである。設計作業について前述した際、スカラー倍演算に対し、行列の各要素の演算ごとにタスクを生成する例を挙げた。この例に限って言えば、様々な理由でそのような並列化を決して行ってはいけない：

- 行列の各要素のタスクをコアの１つに割り当てる方法は、単純に実際の大きさの行列へ拡張できないだろう[x]。
- 並列化によってもたらされるオーバヘッドによって逐次コードの性能を超えないだろう。

このことは、適切な処理量（または粒度）を決定することが自明でないことを強調している。ソフトウェアエンジニアは、初めてこの作業をする際、アプリケーションを正しく並列化できた後、試行錯誤することになるだろう。粒度見積に役立つヒューリスティックがあり、使用するターゲットプラットフォーム、並列化ライブラリ、並列化に使用される言語に対し利用可能なものがあるかもしれない。例えば、Thread Building Blocks[xi]では、“grainsize”（タスク本体で実行する命令数の粒度に関する指標）が 10,000 から 100,000 命令であることを推奨している。

ソースコードの並列化を行った後、高速化によって期待した結果が得られること、またアプリケーションの回帰テストに合格していることを確認するために評価ループ（Figure 2）に立ち戻る必要がある。

## 3.5 High-Level Design

アルゴリズムの意図を理解し、プログラムを逐次実装した後は、アプリケーションの並列実装のための高レベル設計を始めるのがよい。高レベル設計の目的は、アプリケーションを、並列動作する処理に対応した別々の領域(region、訳者注：コード中のある範囲のこと)へと分割することである。領域間の依存関係は並列性の制約となる。加えて、実行時間に関するターゲットの特徴も、実際問題として並列動作が可能な作業の量に制約をかける。並列実行環境への効率的なマッピングのため、分割の粒度は適切に選ばなければならない。

### 3.5.1 Task Parallel Decomposition

タスク並列分割は異なるタスクを並列に動作させることを可能にする。それらの異なるタスクは共有データを読むかもしれないが、独立した結果を作り出す。画像認識アプリケーション例(Figure 5)内では、4つにわけられた認識(Identify)タスクが同じ入力画像データを共有している。各々のタスクは画像内の異なるオブジェクトを認識するようになっている。4つのタスク全ての動作が終われば、それらの集約データを下流工程で扱う。

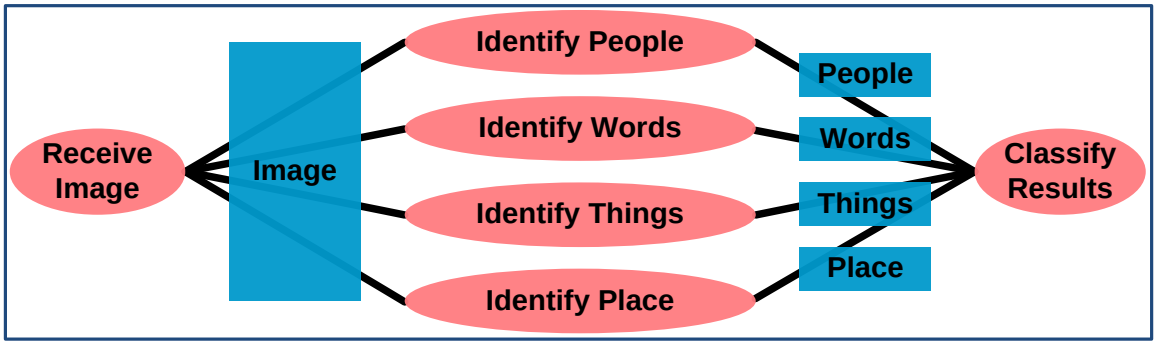



**Figure 5.Task parallel image processing example.**

ある実行プラットフォームにおいて、異なる「認識」タスクは異なる実行時間となる場合がある。4つのタスクはそれぞれ並列に実行できるが、もし「言葉(Words)」「物(Things)」「場所(Place)」の認識がそれぞれ「人物(People)」の認識より3倍速いとすると、「言葉」「物」「場所」の認識タスクを順次実行し、それと並列してより遅い「人物」認識タスクを動作させることは理にかなっている。この種の負荷分散は、利用可能な計算資源を最も効果的に活用するため、タスク並列分割の適切なスケジューリングがいかに重要か示す例となっている。

### 3.5.2 Data Parallel Decomposition

データ並列分割は、入力データを独立した領域に分割し、異なるデータ領域に対する同じタスクを並列に実行させることに焦点を当てたものである。画像認識アプリケーションに関する別の例(Figure 6)では、出力画像の各々の画素は入力画像のいくつかの画素の組み合わせから計算される。ある出力画素の計算はそれに隣接する画素の計算と入力画素を共有するかもしれないが、それぞれの出力画素は他の出力画素とは独立に計算できる。

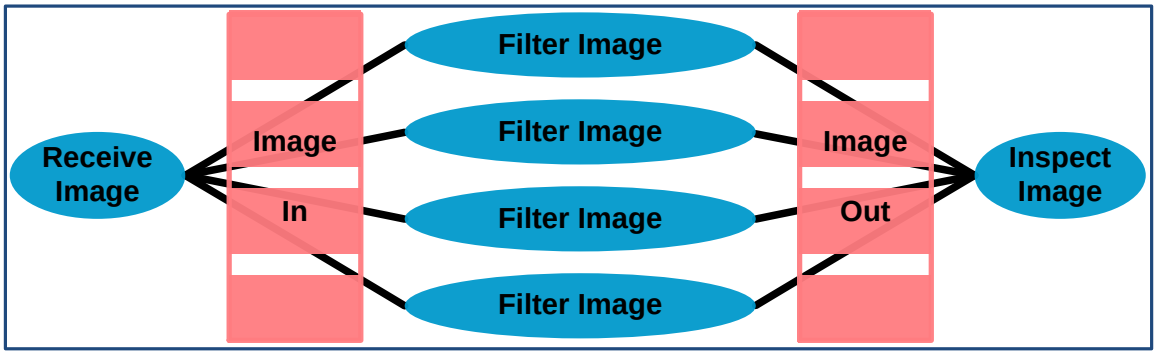



**Figure 6. Data parallel image processing example.**

それぞれの画素は画素ごとに並列に計算できるが、分割の粒度は利用可能な計算資源に合わせなければならない。例えば、もし 8 個の論理プロセッサコアが利用可能であった場合、画像は 8 つの独立した領域へと分割することになろう。領域内では画素は逐次的に計算されるが、領域同士は完全に並列に計算される。データ分割は本質的に自然な負荷分散をもたらす。タスクが画素あたり同じ時間で計算する時（これは静的な仕事量を持つことを表す)、負荷を均等にするためにデータを等分割するのがよい。

### 3.5.3 Pipelined Decomposition

パイプライン分割はタスク並列分割、データ並列分割のハイブリッド型と考えられ、各々のタスクは独立した複数の機能ステージに分割される。それぞれのステージはデータの一部を処理し、その結果を次のステージへと渡す。それぞれのステージは逐次的にデータを処理するが、全てのステージは並列に動作してスループットを増加させる。

画像エッジ検出(edge detection)アプリケーションの例では、画素補正(Correct)、平滑化(Smooth)、ソーベルエッジ検出[xii](Sobel)のステージから構成される(Figure 7)。パイプライン分割を用いると、ある画素ブロックは、パイプライン上で、画素補正ステージから入力され、ソーベルステージへと流される。データが最初のステージから次のステージへ移動する時、次のデータが最初のステージへと入力される。それぞれの処理単位が、アルゴリズム内の異

なるステージに存在する異なるデータブロックを処理することで、3つのステージ全てを並列に実行できる。

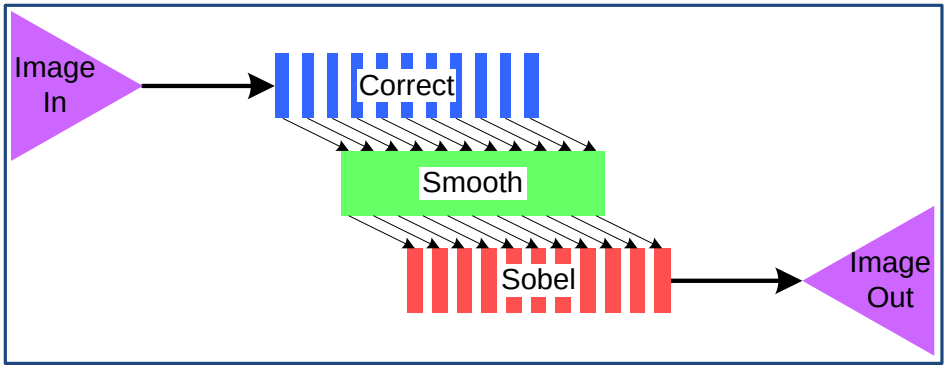



**Figure 7. Pipelined edge detection**

理想的には、パイプライン分割は、利用可能な計算資源とステージ数が同数であることが望ましい。また、パイプライン全体のスループットの上限は最も遅いパイプラインステージで決まってしまう。上記の例では、平滑関数が最も多くの仕事量を要し、各々のブロックに対しては、平滑関数が完了するまで補正関数とソーベル関数がアイドル状態とならなければならない。そのため平滑関数の実行速度が並列実装全体の性能を決めている。平滑関数を並列化することは、それ自身の実行時間をより短くするとともに、パイプラインのバランスもより良いものにする。

### 3.5.4 SIMD Processing

現代のプロセッサの殆どは、SIMD (Single Instruction, Multiple Data)命令と言われる、ある種のベクトル命令を持っている。SIMD 命令により、複数のデータ要素に対して同じ命令を並列に適用することができる。例えば、4組の 8bit 値の掛け算をし、結果として 4 つの 16bit 値を保存する命令などである。この種の低レベル並列性は、1ループあたりの仕事量を増やすため、ループ内で用いられる。可能ならば、この命令レベルの並列性を先に適用し、先に述べたタスクレベルのデータ分割を次に行うべきである。

### 3.5.5 Data Dependencies

アルゴリズムが逐次で実装された時、明確に定義された命令順序が存在し、それはあまりにも柔軟性がないことがある。エッジ検出の例では、与えられたデータブロックに対し、平滑関数が完了するまでソーベル関数を計算できない。しかしながら、他の計算の組み合わせに対しては、例えば補正関数内では、画素を補正する順序は意味がないだろう。

データ読み書き間の依存性は計算の半順序を決定する。順序制約となるデータ依存には 3 つの種類があり、真のデータ依存(true data dependency)、逆依存(anti-dependency)、出力依存(output dependency)とよばれる (Figure 8) 。

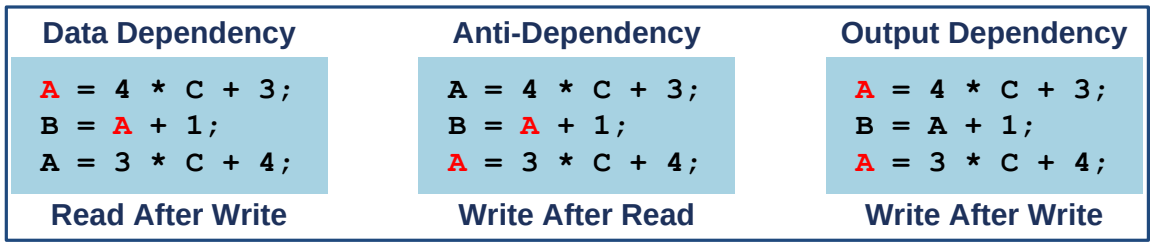



**Figure 8. Three types of data dependency examples (the dependencies are highlighted in red): a) data dependency; b) anti-dependency; and c) output dependency.**

真のデータ依存は、あるデータ値への書き込みが終わるまでは読み込みができないような処理間の順序を示す。これはアルゴリズム内の本質的な依存であるが、このデータ依存性の影響を最小化するようアルゴリズムを改良することができる場合もある。

逆依存は真の依存とは反対の関係を持ち、変数名変更することで解決しうるものである。逆依存では、前のデータが読み込まれるまで新しいデータを書き込むことができない。Figure 8b では、最後の A への代入は B が A の前回値を必要とするため B が代入されるより前に実行できない。最後の式において A の名前を D に変更することで、B と D の代入順序を入れ替えることができる。

変数名変更によって新しい変数を用意し、並列化によって新旧変数の生存期間が重なるとき、メモリ量を増やすことになる場合がある。逆依存は逐次コード内でよく発生するものである。例えば、ループの外側で定義した中間変数を各ループ内で用いることがある。このことは、処理を逐次的に実行するならば正しい。その変数に対する同じメモリ領域を何度も再利用することが出来る。しかし、共有メモリを用いる場合、もし全てのループが並列に動作したとすると、それらは中間変数に対する同じ共有メモリ領域で競合することになるだろう。一つの解法は、ループごとにローカルな中間変数を用いることである。適切なスコープでの変数宣言によって変数の生存期間を最小化することは、この種の依存性を回避することに役立つ。

第三の依存性は出力依存である。出力依存では、もしある変数に対する複数の書き込み命令が完了した際、最後に残る変数値が変わってしまう場合に、それらの書き込み命令は入れ替えられない。Figure 8c では、最後に残る値が不正になるため、最後の A への代入は最初の代入より前に実行することができない。

アルゴリズムを並列化することは、依存性に気を遣うことと、利用可能な計算資源に並列性を適切に合わせることの両方が必要である。多くのデータ依存性を伴うアルゴリズムは効果的な並列化ができないだろう。全ての逆依存性を取り除いてもなお分割によって十分な性能が得られない場合、同じ結果を得られる、より並列化しやすいアルゴリズムへ変更することを考えなければならない。このことは、厳密に定められたアルゴリズムを用いた標準仕様の実装では不可能かもしれないが、それ以外の場合には、同様の結果を得られる効果的な方法が存在するかもしれない。

## 3.6 Communication and Synchronization

データ依存関係によって、正しい動作を保証するために守らなければならない順序要件が定められる。順序を保証するためには、それぞれの並列プログラミングモデルにおいて、異なる手法を使う必要がある。

### 3.6.1 Shared Memory

共有メモリシステムでは、それぞれのプロセッサは個々にメモリをキャッシュしているかもしれないが、システムはプロセッサ間の共有メモリの一貫性を維持しなければならない (Figure 9)。キャッシュにおいて、プロセッサ間でキャッシュラインを共有しようとすると、メモリの一貫性を維持するためのシリコンコストは非常に高くなる。メモリ内の適切なデータ配置により、このペナルティを最小化できる。

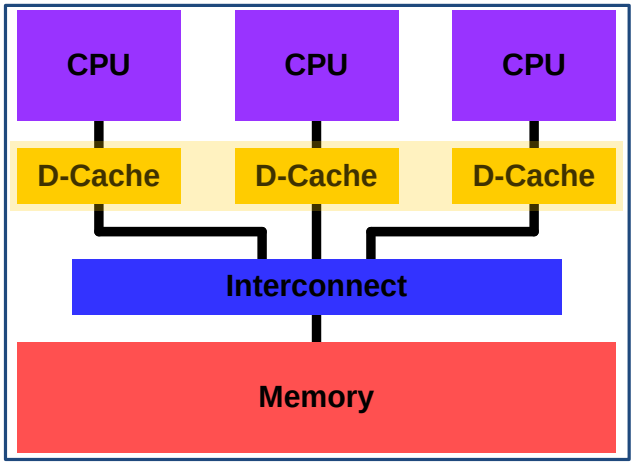



**Figure 9. Processing units share the same data memory.**

マルチスレッドは、同時に複数のタスクを実行するために使われる。タスク間で共有されるデータに対しても、依存関係が正しく守られるよう、開発者はアプリケーション内で適切に

同期(つまりロック)しなければならない。例えばスレッド T0 と T1 がどちらも変数 A を読み書きする必要があるとする(Figure 10)。同期なしでは 2 つのスレッド間の読み書き順序は予測できず、3 つの異なる結果になりうる。

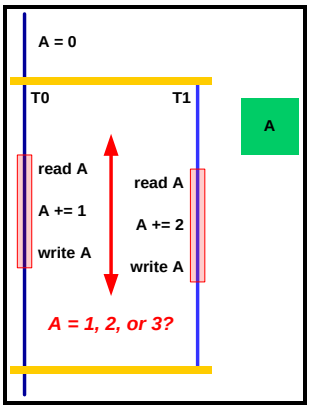



**Figure 10. Shared data synchronization.**

ロックやセマフォを利用した排他制御は、あるクリティカルな依存関係を含むコード（クリティカルセクション）を一度に 1 つのスレッドのみが実行できることを保証した一般的な技術である。あるスレッドがクリティカルセクションのロックを保持している間は、他のスレッドが入ってくるのをブロックする。もしそれを許すと、セクション間共有データが関わる依存関係を守れない可能性が出てきてしまう。Figure 10 の例では、クリティカルセクションのロックを適切に行うことにより、A の最終的な値が常に 3 になることを保証できる。

一般に、各プロセッサは一度に 1 つのスレッドを実行する。プロセッサ全体では、プロセッサ数より多くのスレッドが生成されるかもしれない。あるスレッドが、ロックが解放されるのを待つ間ブロックされるとき、オペレーティングシステムは、実行の準備ができている別のスレッドを起動させるだろう。

ロックの取得と解除のコストが重要になる。ロックはコードを逐次化するので、大きなクリティカルセクションをロックすることは並列処理を妨げる。一方で、低レベルロックを頻繁に使用すると、同期のために大きなペナルティが課される。タスク生成や完了したタスクの破棄のコストもまた重要であるので、タスクとロックの粒度は利用可能な計算資源に一致させなければならない。

### 3.6.2 Distributed Memory

分散メモリシステムでは、メモリはシステム間で共有されず、各プロセッサは自身のローカルメモリを管理する(Figure 11)。異なるプロセッサ上で実行中のタスク間でのデータ通信は、それらの間でデータを送受信することによって行われる。これは、メッセージパッシングと

呼ばれることが多い。スレッドプログラミングモデルは共有メモリにおいて適しているのに対し、メッセージパッシングモデルは分散メモリと共有メモリのどちらにも使用することができる。

分散メモリモデルでは、データは「明示的に」タスク間で共有される必要がある。タスク間の同期は、同期送受信セマンティクスによって達成される。受信側タスクは送られるデータが利用可能になるまでブロックされる。一方、データが利用可能になった時、受信側タスクがブロックすることなく確認できるまたは通知される場合には、非同期送受信セマンティクスが利用できる。これにより、通信と計算のオーバーラップが可能になるため、大幅な性能向上につながる。

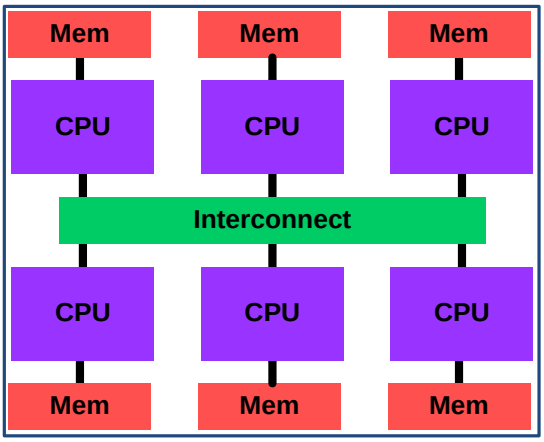



**Figure 11. Distributed memory architecture.**

共有メモリと比較して、分散メモリは、メッセージの生成や破棄、明示的なデータコピーによって高い通信オーバーヘッドを示すので、メッセージパッシングはこれらの機能に関して最適化する必要がある。

## 3.7 Load Balancing

計算あたりの仕事量は固定であるか、もしくは入力データ依存である。仕事量が固定している場合、静的ワークロードと呼ばれ、設計時に分割されることもある。先に述べたエッジ検出では、仕事量はすべての画素において同じであり、データ分割により並列性は利用可能なプロセッサ全体に容易にスケールする。

平滑関数の実行時間が入力データに依存していたら、何が起こるだろうか。例えば、画素値がゆっくり変化する領域では平滑化するためにより少ない計算サイクルですむことがある一方で、急速に変化する画素値を持つ領域は、追加の処理が必要になることがある。このよう

な場合の仕事量は、画素値に依存して動的に変化する。すなわち、同じ大きさの画素領域でも仕事量が大きく異なる場合がある。例えば、上半分が大きく変化し、下半分が単色の画像を考えてみる。もしも、これを上下二つのタスクに分割し、異なるプロセッサで実行することを考えると、下半分は上半分よりも先に完了し、下半分を受け持つプロセッサは待機状態になるだろう。

仕事量が予測できないとき、スケジュールは統計的にしか最適化できない。この場合、小さな領域に仕事を分割することは、領域間での仕事量の差異が小さくなるので、処理効率が向上する傾向がある。しかし、仕事が小さなタスクに分割されると、通信コストがタスク実行時間の大部分となることがあるため、通信コストを考慮して最適なタスクサイズを決めなければならない。スレッドプールやワークスチールを使用する実行環境には、動的負荷分散が組み込まれている。

タスク、データ、およびパイプライン分割においては、分割領域間での仕事量均衡について常に検討する必要がある。静的ワークロードは解析が容易であるが、動的ワークロードでは、利用可能な計算資源と一致し、かつ仕事量との不一致を軽減するような、適切な粒度を見つけるために統計的解析が必要になるだろう。

## 3.8 Decomposition Approaches

高レベル設計プロセスでは、データ依存要件、通信要件、同期要件を考慮しながら並列処理領域へとアルゴリズムを分割する。またそのとき、実装時での効率的分割を達成するために負荷均衡を考える。

### 3.8.1 Top-Down or Bottoms-Up

アルゴリズム分割においては、2 つの主要な方法がある。トップダウンとボトムアップである。トップダウン手法では、アルゴリズムは独立した領域に分割される。逐次のまま残る領域がある一方、タスク、データ、およびパイプライン分割を使用して分割できる領域もあるかもしれない。最も効果がある領域分割を見つけるために、高レベル解析でのプロファイリング情報を使用すべきである。各々の分割後、並列性能を見積もることができる。十分な性能が達成されない場合、さらなる並列性を見つけるため領域の細分化を試みることもある。

一部の領域は容易に多くの細粒度領域に分解されるかもしれないが、粒度とオーバーヘッドとのトレードオフが存在する。画像をピクセルにデータ分割する方法は、細粒度実装の極端な例かもしれない。要求されるコア数、タスクの作成と破棄のオーバーヘッド、および通信と同期のオーバーヘッドはかなりのコストになる。

先に説明したように、高レベル解析でのプロファイリング情報を用いて最も重要なホットスポット領域を特定する。最もよい結果を得るために、実行時間を比較的多く消費するホットスポットの並列化に注力すべきである。そのとき、真のボトムアップ手法は、ホットスポット領域を最も細粒度に分割するところから開始し、分割とオーバーヘッドとのバランスを考えながら、より大きい領域へと徐々にまとめていく方法である。一旦適切な粒度が確立されると、推定性能レベルに十分達するまで、小さな領域を並列に組合せながら大きな領域を作り上げていくことになる。

### 3.8.2 Hybrid Decomposition

分割は、計算資源の利用可能性や種類を考えながら、階層的に適用される。例えば、第 1 ステージが第 2 ステージより 6 倍遅いような、2 ステージパイプライン分割を考えてみる（Figure 12）。当初のパイプライン分割では、より遅い第 1 ステージばかり実行することになるだろう（図の上部）。

データ分割可能性と十分な計算資源を仮定することにより、第 1 ステージでは 6 個のデータブロックを並列に実行し、その後、同じ処理量で第 2 ステージの 6 個のデータブロックを逐次的に実行することができる（図の下部）。これにより効果的にパイプラインの均衡をとり、スループットを最適化できる。

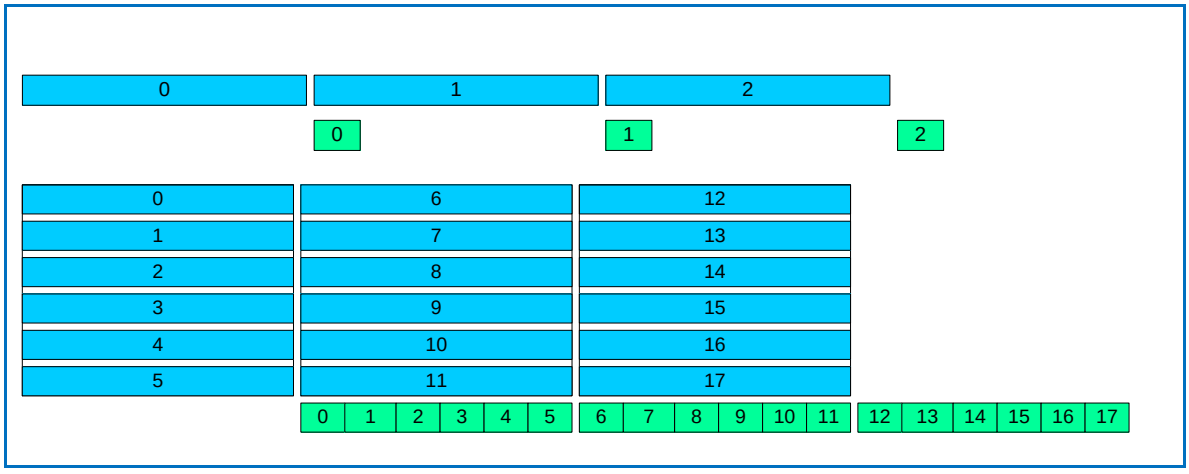



**Figure 12.Example depicts 2-stage pipeline combining data and pipeline decomposition.**

# CHAPTER 4: IMPLEMENTATION AND LOW-LEVEL DESIGN

## 4.1 Introduction

並列実装はプラットフォームアーキテクチャに強く依存する。本章では代表的ないくつかのアーキテクチャに対して（スレッド化やメッセージパッシングのような）並列実装技術や（段階的修正や逐次的等価性のような）推奨される実装手法について述べる。よい実装は高レベル設計指標を満たし、デバッグを容易にし、潜在的なバグを減らす。

## 4.2 Thread Based Implementations

スレッドはソフトウェアプログラムにおいて並列化を実現する基本的な手段である。軽量プロセスのように見え、オペレーティングシステムが特定の機構を介して生成、管理、破棄を行う。その機構とは独立して、スレッドを使用する目的は、アプリケーションの応答性向上、複数プロセッサを搭載した計算機の効率的利用、プログラム構造や効率の改善である。歴史的に、スレッド環境の多くの実装は一般的に特定のプラットフォームやユーザからの要求への対応が中心となっている。基本的なスレッドモデルとして、ユーザレベルスレッド（User Level Thread (ULT)）とカーネルレベルスレッド（Kernel Level Thread (KLT)）がある。

ユーザレベルスレッドに関して、カーネルはプロセスの管理をするが、ユーザレベルスレッドを認識することはできないので、ユーザがスケジューリングや管理を行う責任がある。しかし、もし ULT がカーネルを呼び出したならば、全体のプロセスがブロックされる。カーネルレベルスレッドの場合、カーネルが各スレッドのスケジューリングや情報を管理しなければならない。プロセスのブロックはスレッド毎に行われる。ハイブリッド ULT/KLT スレッドは Linux や Solaris のようなプラットフォームで実装されている（Figure 13）。

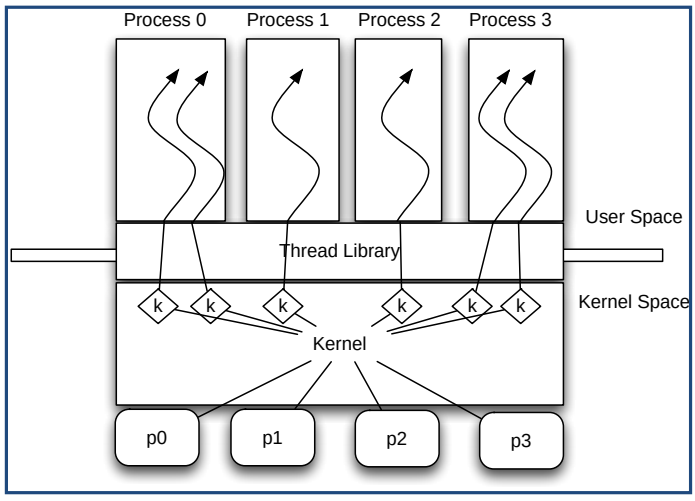



**Figure 13. Hybrid threading model.**

スレッドの生成や管理は、生成、破棄、同期、スケジューリング、プロセス間相互作用に関する関数呼び出しによって行われる。

他の基本的なスレッドの特性は次の通りである：

- どのスレッドから生成されたのかはわからない。
- 同じプロセス内の全てのスレッドは同じアドレス空間を共有する。
- 同じプロセス内のスレッドは命令、データ、ファイルディスクリプタ、シグナルおよびシグナルハンドラを共有する。
- 各スレッドは固有のスレッドID、レジスタとスタックポインタ、ローカル変数のためのスタック、リターンアドレス、シグナルマスク、優先度とその戻り値を持つ。

## 4.3. Kernel Scheduling

カーネルは内部のスケジューリングアルゴリズムに従って、ある計算機上のプロセスやスレッドのスケジューリングを行うようプログラミングされている。カーネルによるスレッドのスケジューリングには以下のような特徴がある：

- カーネルはスケジューリングアルゴリズムや計算機の状態に基づき、スレッドの制御を切り替える。
- カーネルはスレッドのコンテキストを保存、復元する（コンテキストスイッチとして知られている）。これにはオーバーヘッドがあるため、ソフトウェア性能に悪影響を及ぼす上、マイクロプロセッサの動作やキャッシュ状態にも悪影響を与える。
- カーネルのスケジューリングは非決定的であり、いつも同じ順序でスケジューリングされるわけではない。
- プログラマはいくらかのスケジューリング制御が可能ではあるが、適切なアルゴリズムを設計し、入力データや実行条件を変えてその性能を確認することが重要である。

## 4.4 About Pthreads

POSIX スレッド（Pthreads）はスレッドを生成、操作するための API である。この標準では、オプションのコンポーネントや実装依存セマンティクスを含む、およそ 100 の関数が定義されている。この API の実装は Solaris、MAC OS X、HP-UX、FreeBSD、GNU/Linux など、ほとんどの POSIX 準拠のプラットフォームで利用可能である。Windows においても、pthreads-

w32 プロジェクトの貢献により、API の一部[xiii]を使うことができる。API は通常、ユーザとカーネルの間の層を形成し、ライブラリとして提供される。この MPP ガイドでは C++言語の例を紹介するが、Pthreads ライブラリは C や Fortran をはじめ、その他多くの言語で使用できる。

たとえ POSIX 準拠のプラットフォームが Pthreads を実装していても、いくつかの実装が不完全であったり、仕様に準拠していなかったりすることがある。いくつかの API において、下にある OS カーネルの対応がない、プラットフォームのライブラリに存在しないといった可能性があるためである。スレッド化ソフトウェアを開発する時はプラットフォーム固有の文献を確認してほしい。

Pthreads に関しては、この本より深く述べられている古典的な参考書[xiv] [xv] [xvi]がある上、多くのオンラインチュートリアルも存在する。

## 4.5 Using Pthreads

C や C++で書かれる Pthreads プログラムでは、API 関数を使うために pthread ヘッダファイルをインクルードしなければならない (*#include <pthread .h>*)。ほとんどの Unix 系プラットフォームでは、ライブラリとユーザバイナリファイルをリンクさせるために'*-lpthread*'を使う。

各 Pthreads 関数は、関数パラメタとして不透明（opaque）型の変数（またはオブジェクト）を用いる。スレッドライブラリはそのオブジェクト/リソースの状態を追跡するためにこのパラメタを用いる。例えば *pthread_create* 関数では、もろもろの情報とともにスレッド ID を含む *pthread_t* 型が使用される。この ID は *pthread_self* 関数呼び出しによって取得できる。

ほとんどの Pthreads 関数では成功、失敗または呼び出しの状態を示す値が返される。これらの値をチェックし、適切に対処する必要がある。

Pthreads 関数の分類について、Table 1 に概要をまとめた。以降ではそのほとんどの API について紹介し、その適用について説明する。

| Function prefix | Function |
|---|---|
| pthread_ | Thread Management |
| pthread_attr_ | Thread attributes |
| pthread_mutex_ | Mutexes |
| pthread_mutexattr_ | Mutex Attributes |
| pthread_cond_ | Condition variable |
| pthread_condattr_ | Condition variable attributes |
| pthread_key_ | Thread specific data |
| pthread_rwlock_ | read/write locks |
| pthread_barrier_ | Barriers |

**Table 1. Pthreads functions.**

## 4.6 Dealing with Thread Safety

コードをスレッドセーフにするために（訳者注：スレッド化して並行動作させても不具合を起こさないコードにするために）、複数のスレッドが共有資源にアクセスすることを防ぐための特別な対策をする必要がある[xvii]。対策としては、a) 全てのアクセスが共有資源を書き換えない、b) 全てのアクセスが冪等性（べきとうせい、idempotent）をもつ、すなわち実行順序が結果に影響を与えない、c) 一度に 1 つのアクセスのみ許可する、といった方法が挙げられる。

複数の実行スレッドにアクセスされる共有メモリは、1 つ以外の全てのスレッドを排他制御する目的で、書き込みに対する同期機構によって保護される必要がある。複数のスレッドが（見かけ上）同時に 1 つの共有資源にアクセスし、1 つの書き込みだけある場合、結果は各スレッドの相対的な実行順序に依存するため、競合状態が発生する。

同期を行い、排他制御によって保護しなければならないコードの部分はクリティカルセクションと呼ばれる。そして、全てのスレッドがクリティカルセクションの前後で entry ルーチンと exit ルーチンを実行するような、保護機構（Figure 14）の構築が必要である。

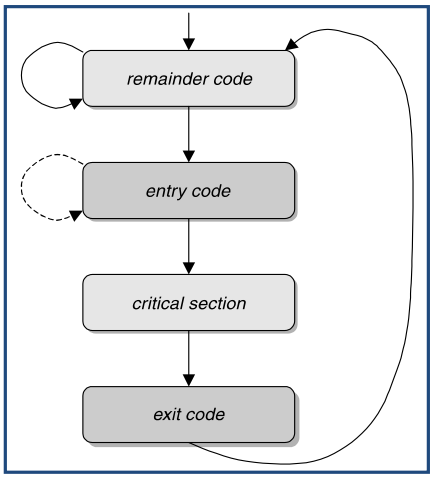



**Figure 14. Critical section protection requirements.**

Pthreads においては、保護されないグローバル変数と静的(static)変数は特に厄介であり、使用を避けるべきである。スタック上にメモリを確保したスレッドは、独占利用できるスレッドスタック空間を、自動的に活用することができる。多くの関数は静的データへのポインタを返すが、それが問題となる。ヒープ上にメモリを確保し、そのメモリへのポインタを返すことにより、この問題は改善できる。スレッドセーフ版ライブラリ関数の使用も推奨されている。なお、サードパーティ製ライブラリを使う場合にはスレッドセーフであるかについて吟味する必要があり、スレッドセーフな関数を使うべきである。

## 4.7 Implementing Synchronizations and Mutual Exclusion

排他制御は、どの二つのスレッドも同時に同じクリティカルセクションに入らないように行われる。その上で、デッドロックやスタベーションが発生してはいけないことも意味している (Table 2[xviii])。この問題を解くために、初期には、ダイクストラやデッカーのアルゴリズム[xix xx xxi]のような多くの試みがあったが、その後の研究により[xxii xxiii xxiv]、今ではピーターソンのアルゴリズム[xxv]が最も普及している (ミューテックスアルゴリズムに関するいくつかの興味深い迷信は存在する[xxvi]) 。

| | |
|---|---|
| 排他制御 | 二つのプロセスが、クリティカルセクションを同時に実行しないこと |
| デッドロックフリー | あるプロセスがクリティカルセクションに入ろうとしたときに、同じプロセスあるいは別のプロセスが、そのクリティカルセクションにいつかは入ること |
| スタベーションフリー | あるプロセスがクリティカルセクションに入ろうとしたとき、そのプロセスがいつかはそのクリティカルセクションに入れること |

**Table 2. Mutual Exclusion Properties.**

通常の処理において、スレッドが利用資源の競合を起こす場合には、資源を共有したり、何らかのスレッド間通信手法を通してやりとりしたりすることにより、期待する結果を保証している。そのようなスレッド間相互関係の中で、プログラマが、競合状態、デッドロック、スタベーションの犠牲になる状況は数多く存在する。

デッドロック[xxvii]は、二つのスレッドが、両方とも相手が必要としている共有資源を保持し続けつつ、相手の保持している共有資源の解放を待っている場合に発生する状況である。しかし、外部からの干渉なしに共有資源を解放することはなく、循環待ち(circular waiting)と呼ばれる状況に行き着く。デッドロックは、ロック順を変更することで通常防ぐことができる。ロック順とはロックが取得・解放される順番である。例として、昇順にロックを取得し、降順にロックを解放するという経験則がある。C++を使用している場合 RAII( Resource Acquisition Is Initialization（共有資源の確保は初期化時に）)、レベル付けされたロック、ロックガード・スコープロックのような概念がロックの見逃しを防ぐために役立つ[xxviii]。Figure 15 は C++の一般的なスコープロック手法を表している。オブジェクトがスコープの外に出た場合、そのオブジェクトは破壊される。そのオブジェクトのデストラクタ内でスタックが巻き戻される際に、`ScopeLock` クラスのデストラクタがミューテックス(mutex)アンロックを呼び出すことで、そのオブジェクトが破壊された時にミューテックスがアンロックされていることを保証する。

```
class ScopeLock {
private:
  pthread_mutex_t &lock_;

public:
  ScopeLock(pthread_mutex_t &lock) : lock_(lock) {
    pthread_mutex_lock(&lock_);
  }
  ~ScopeLock() {
    pthread_mutex_unlock(&lock_);
  }
};
```

**Figure 15. A common C++ scope-locking technique.**

スタベーションは、多くのスレッドが競合しているクリティカルセクションで排他制御を実行しようとした際に発生する。カーネルは二つのスレッドのみを交互にスケジューリングし、第三のスレッドが共有資源を無期限に待ち続けることがある。第三のスレッドは、共有資源を待ち続けることで、まったく動作しなくなる。これをスタベーション（餓死）という。正しいロックの目標の 1 つは公平であること、それによりスタベーションを防ぐことである。

## 4.8 Mutex, Locks, Nested Locks

Pthreads を使用しているとき、特定の Pthreads 関数を使用することで、同期と排他制御を実装できる。ミューテックスは、効果的にクリティカルセクションを保護する一方で、適切に実装しなければ、依然としてデッドロックやスタベーションを引き起こす。プログラマは、すべての共有資源使用がミューテックスによって保護されていることを保証する責任がある。Pthreads では、同時にミューテックスオブジェクトをロックできるスレッドは唯一つである。ミューテックスを保持しているスレッドがそれを解放するまでは、いかなるスレッドがミューテックスオブジェクトをロックしようとしてもブロックされる。ミューテックスが解放されると、スレッドのうちの一つがクリティカルセクションに入ることができるが、これは必ずしも最初にクリティカルセクションに到達したスレッドとは限らない。

クリティカルセクションに関する経験則:

- 時間：クリティカルセクションはできる限り短時間で終了するようにする。一部の命令は、より長い実行時間を要する。
- 空間：ロックの間は最小限の命令で実行する。
- 非決定的に実行されるいかなるコードもロックしてはならない。
- データが保持されているコンテナを扱う際は、粒度との関係を考慮して、可能な限りコンテナ全体でなくデータ項目や構造体をロックする。

## 4.9 Using a Mutex

ミューテックスを使うには最初に初期化を行い、使い終わったら破棄しなければならない。この初期化と破棄の考え方は Pthreads(Figure 16)の様々な箇所で共通的に使われている。

```
#include <pthread.h>
....

pthread_mutex_t mutex;
int global;

main()
{
pthread_mutex_init(&mutex,NULL); // dynamic initialization

// some code to create and join threads

pthread_mutex_destroy(&mutex);
}

void thread_one()
{
// some code for thread one
pthread_mutex_lock(&mutex);
++global;
pthread_mutex_unlock(&mutex);
// some other code for thread one
}

void thread_two()
{
// some code for thread two
pthread_mutex_lock(&mutex);
--global;
pthread_mutex_unlock(&mutex);
// some other code for thread two
}
```

**Figure 16. A partial example of initializing and locking a critical section with a mutex.**

## 4.10 Condition Variables

状態変数は、変数の変更に基づいて排他制御を実施するための方法である。例えば、２つの状態変数と１つのミューテックスを用い、有限バッファ、共有キュー、ソフトウェア FIFO のいずれかを作ることができる (Figure 17)。C++クラス IntQueue は標準ライブラリである queue オブジェクトをラップしており、ロックの規則を強制的に守らせるために *ScopeLock* クラスを使用している。

```
class IntQueue
{
private:
  pthread_mutex_t mutex_;
  pthread_cond_t more_;
  pthread_cond_t less_;
  std::queue<int> queue_;
  size_t bound_;

public:
  IntQueue(size_t bound) : bound_(bound) {
    pthread_mutex_init(&mutex_, NULL);
    pthread_cond_init(&less_, NULL);
    pthread_cond_init(&more_, NULL);
  }
  ~IntQueue() {
    pthread_mutex_destroy(&mutex_);
    pthread_cond_destroy(&more_);
    pthread_cond_destroy(&less_);
  }
  void enqueue(int val) {
        pthread_mutex_lock(&mutex);
    while(queue_.size() >= bound_)
        pthread_cond_wait(&less_,&mutex_);
    queue_.push(val);
    pthread_cond_signal(&more_);
    pthread_mutex_unlock(&mutex);
  }
  int dequeue() {
        pthread_mutex_lock(&mutex);
    while(queue_.size() == 0) {
      pthread_cond_wait(&more_, &mutex_);
    }
    int ret = queue_.front();
    queue_.pop();
    pthread_cond_signal(&less_);
        pthread_mutex_unlock(&mutex);
    return ret;
  }
  int size() {
    ScopeLock lock(mutex_);
    return queue_.size();
  }
};
```

**Figure 17. Simple concurrent queue using two condition variables and a mutex.**

## 4.11 Levels of Granularity

粒度とは並列プログラムにおける、通信に対する計算の割合である。プログラムには通常、計算を行う部分と通信を行う部分がある。ここでいう通信とは、スレッドの制御、ロック、破棄にかかわるすべてのことを意味する。並列計算における粒度には二つの種類、細粒度と粗粒度、が存在する。

細粒度並列性とは、実行される個々の仕事量が少ない、もしくは通信に比べて計算の割合が小さい場合である。このような場合、負荷分散は容易になるが、スレッドに関わる様々な事柄の調整や、自スレッドの結果を他スレッドに送るようなことに比較的多くの時間を費やすことになる。カーネルが、ミューテックスまたは状態変数を扱うことにプログラム実行時間の大部分を費やしてしまうような状況を作り出す可能性もある。

粗粒度並列性は、計算の割合が通信に比べて高いことである。このことは、性能が向上するかもしれないことを示しているが、アルゴリズムの実装方法によっては、与えられた計算機上で妥当な負荷均衡を実現するために多くの努力が必要になるかもしれない。

適切な粒度を決定するために役に立つ経験則として、計算時間によって通信コストを「償却」するのに十分な量の仕事がなければならないというものがある。ここで、粗粒度から始めて、細粒度並列性を実装する箇所を探すことを考える。通常、並列性は高いほど良いが、それだけで性能を決定付ける要因ではない。適切な粒度は、利用可能な計算資源に加え、スケジューリング戦略、アルゴリズム実装、計算と通信の割合に依存している[xxix]。最大性能を実現する唯一の方法は、多くの実験と性能チューニングを施すことである。さらに、並行処理の数は、通信帯域不足を避けるために、計算機のコア数に近い数であるべきである。

Pthreads 関数が使われる時にはほとんど、ライブラリはタスク実行のためにカーネルに処理依頼をするが、今度はそれが一連の性能問題を引き起こす。そのため、Pthreads 関数の相対的なオーバーヘッドに関して知っておくべきである。ソフトウェアの潜在的並列性に関する知見 (Amdahl's/Gustavson の法則) [xxx]と組み合わさり、最適化プロセスの手がかりとなるだろう。

## 4.12 Implementing Task Parallelism

タスク並列は、それぞれのタスクが異なるスレッドで実行され、それらのスレッドが並列計算機上の異なるプロセッサまたはノードに割り当てられることを意味している。スレッドは、プログラム動作中にデータをやり取りすることがあるため、他のスレッドと通信を行うかもしれない。異なるタスクに対してアルゴリズムが制御やデータフローの依存を課している可能性があり、これによりタスクは実行順序に制約をかけられ並列化は制限される。

Pthreads によるタスク並列化の実装：

- プログラム上で関数群を作成もしくは指定する。これらはスレッド関数とよばれ、タスクとして使用される。
- それらのスレッド関数を実行する複数のスレッドを作成し、それらの関数へ必要なデータを渡す。
- 必要に応じて、それらのスレッド関数を待ち合せ（ジョイン）、戻り値を受け取る。
- 最低限、すべてのクリティカルセクションは保護されなければならない。これは、少なくとも1つのスレッドによって変更され，他のスレッドからアクセスされる共有資源に対応する。

## 4.13 Creation and Join

スレッドは `pthread_create` と `pthread_join` の呼び出しにより作成、ジョインされる(「ジョイン」はスレッド間で同期するための一つの手段である)。スレッドがスレッドを呼ぶときに引数として情報を渡す、戻り値として情報を受け取ることが可能で、これによりスレッド間通信が可能である。以下の例(Figure 18)では、スレッドの作成とジョインを例示しており、スレッドにスレッド番号を渡している。少し変更を加えると、スレッド関数内の `pthread_exit` において設定された戻り値を、`pthread_join` の二つ目のパラメタを通して取得することができる。

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM_THREADS 5

void *print_hello( void *threadid )
{
        printf( "\n%d: Hello World!\n", ( int ) threadid );
        pthread_exit( NULL ); // potenitally return some value here
}

int main()
{
        static pthread_t threads[ NUM_THREADS ];
        int rc, t;
        for ( t = 0; t < NUM_THREADS; t++ ) {
                printf( "Creating thread %d\n", t );
                rc = pthread_create( &threads[ t ], NULL, print_hello, ( void * ) t );
                if ( rc ) {
                        printf( "ERROR; pthread_create() returned %d\n", rc );
                        printf( "Error string: \"%s\"\n", strerror( rc ) );
                        exit( -1 );
                }
        }
        for ( t = 0; t < NUM_THREADS; t++ ) {
                printf( "Waiting for thread %d\n", t );
                rc = pthread_join( threads[ t ], NULL); // potentially get a return value here
                if ( rc ) {
                        printf( "ERROR; pthread_join() returned %d\n", rc );
                        printf( "Error string: \"%s\"\n", strerror( rc ) );
                        exit( -1 );
                }
        }
        return 0;
}
```

**Figure 18. Pthreads creation and join.**

## 4.14 Parallel Pipeline Computation

前述のとおり、パイプラインは逐次的に実行されるタスクの集合であり、最初のタスクの入力はあるデータ源から取得され、あるステージの結果は次のステージへ供給される。それぞれのパイプラインステージは一つの並列タスクを表す。パイプラインは三つのフェーズからなる。1) (タスクの)受信、2) (タスク上で行われる処理の)実行、3) (次のステージへタスクを)送信。

並列パイプラインシステムは時間的および空間的データ依存を持っており(Figure 19)、したがってそれぞれのステージの出力時に同期を考える必要があるかもしれない。

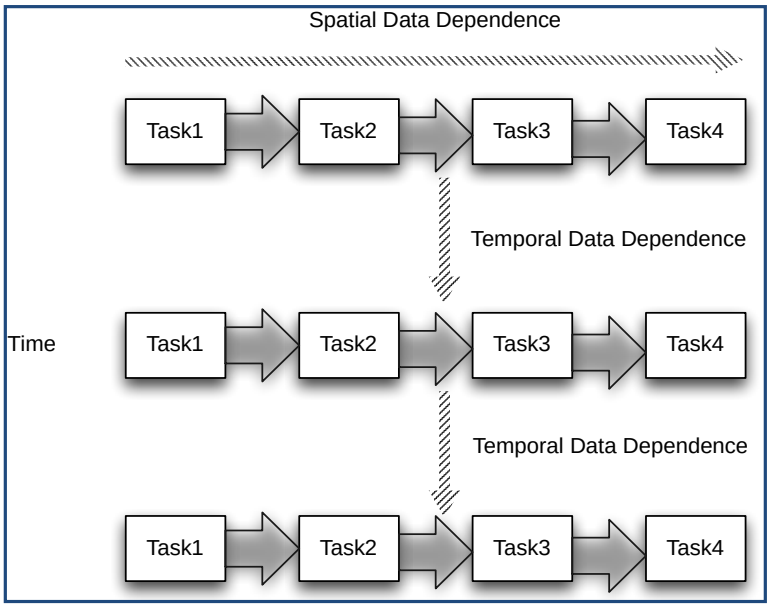



**Figure 19. Parallel pipeline system with temporal and data dependencies.**

パイプラインは以前に導入されたキューの概念から作成でき、そこでは、各パイプラインステージは別々のキューオブジェクトになる。各ステージは、前のキューからデータを取り出し、何らかの処理を行い、次のキューにデータを追加するスレッドを一つ以上持つ。

## 4.15 Master/Worker Scheme

マスター/ワーカー方式は動的タスク分配(Figure 20)を用いたタスクプールに使用できる手法である。マスタースレッドは処理をワーカースレッドに渡し、結果を収集する。この方式では、多対一の通信システムにより、マスターに結果を収集するたびに新しいタスクをワーカーに供給することで均等に処理を分配することを可能にしている。また、負荷均衡に影響がないならば、一つのワーカーに多くのタスクを与えることも可能である。

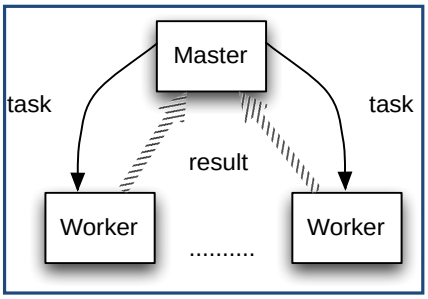



**Figure 20. Master/worker scheme supports dynamic task distribution.**

マスター/ワーカーシステムは、結果を収集しタスクを渡すマススレッドの、比較的スムーズな動作に依存している。多くのワーカーが使用されるとき、マススレッドは過負荷になるかもしれない。逆に、より多くの処理をワーカースレッドに割り当てること (粗粒度アプローチ)はシステムの動的な性質に制約を与え、負荷均衡に影響を与える可能性がある。この問題に対処するため、木の葉をワーカー、中間ノードをサブマスターとした、木構造による階層的なマスター/ワーカースレッドシステムを構成することができる。

## 4.16 Divide and Conquer Scheme

分割統治方式は、多くのソート、計算幾何学、グラフ理論、数値計算問題を解くために使われる重要なアルゴリズムである。分割統治方式には以下の 3 つのフェーズがある。

1) 分割フェーズ - 問題が小さなサイズの一つ以上の独立な部分問題に分割される。
2) 統治フェーズ - それぞれの部分問題が再帰的もしくは直接的に解かれる。
3) 集計フェーズ - それぞれの部分解が元の問題の解として集計される。

分割統治モデルでは、一つ以上のスレッドが並列に同じタスクを実行する(SPMD モデルとなる)。マススレッドはなく、全て独立に実行する。再帰呼び出しは、それぞれの呼び出しがプログラムメモリの異なる部分に書き込む場合のみ、並行して行うことができる。分割統治アルゴリズムでは、均等でない問題を扱う場合に、負荷分散問題が起こる。この問題は、部分問題をさらに小さく分割することができれば、解決できる。

## 4.17 Task Scheduling Considerations

タスク粒度の決定は分割問題である。プログラムは並列実行に適したタスク群に分割されなければならない[xxxi]。「粒度」はタスクとして実行される命令の集合として定義される。言語によっては、これは関数やループ本体になるかもしれない。

並列システムにおいて利用可能な計算資源を効率的に使用するためには、タスクが並列実行可能になるようプロセッサコアに十分なタスクを分散させる必要がある。プログラマは、実行時間を最小化するような、最適なタスクサイズを決定しなければならない。タスクのサイズが大きくなるほど、与えられた計算機での並列性は低くなる。小さいタスクサイズは大きな並列オーバーヘッド（通信と同期）をもたらす。

粒度問題は一般に NP 完全問題ではあるが[xxxii]、部分問題に対して最適に近い解を見つけることは可能である[xxxiii]。また、タスクパッキング[xxxiv xxxv]やワークスチール[xxxvi xxxvii]のような手法は、スケジューリングアルゴリズムの性能向上に有効である。

## 4.18 Thread Pooling

スレッドプールはスレッドの集合であり、各スレッドは関数やタスクを実行時に動的に割り当てられる。タスク数や利用可能なプロセッサ数のようなシステム状態にしばしば依存するが、プログラマはプールの動作やサイズを制御できることがある。プールは、特定の負荷と計算機特性の下で最高性能を達成するよう調整される。この技術は主に、スレッドを再利用し、スレッドを何度も生成、破壊するオーバーヘッドを避けることを可能にしている。

## 4.19 Affinity Scheduling

アルゴリズムやリソース（資源）取得・管理方法、その負荷に応じて、あるスレッドを特定のプロセッサに割り当てること（アフィニティという）が有効なことがある。OS カーネルが、電力やキャッシュ、CPU、その他のリソースにもとづいたスレッドスケジューリングをすることがある。例えば OS カーネルが、キャッシュ利用にもとづき(すなわちキャッシュをフラッシュして書き換えることを避けるために)、プロセッサ上での連続実行の有効性を検出し、スケジュールする場合がある。このキャッシュを考慮した手法はキャッシュアフィニティースケジューリングとして知られている。

ほとんどのプロセッサアーキテクチャにおいて、キャッシュやメモリ、プロセッサ境界を越えてのスレッド移行は、処理時間が長く（TLB フラッシュやキャッシュ無効化など）、プログラム性能を低下させる傾向がある。プログラマはスレッドアフィニティを設定することにより、キャッシュや割り込み処理共有をうまく活かしたり、計算をデータ（局所性）に整合させたりできる。さらに、一つの CPU にアフィニティを設けて OS が割り当てるリソース範囲外とし、他のスレッドがその CPU を使えないようにすることで、一つのスレッドがその CPU を専有することができる。アフィニティスケジューリングを手動により制御する場合には、確実に効果をあげるために、プログラマはアフィニティを注意深く設計すべきである。

アフィニティ設定、つまりスレッドとプロセッサとの結合、のための関数は OS によって異なる。Linux の場合には *sched_setaffinity* や *pthread_setaffinity_np* (np とは non-portable の略)が用いられ、Solaris の場合には *processor_bind* が用いられる。どちらも同じような意味合いを持つ。

以下の例(Figure 21)は、キャッシュアフィニティスケジューリングのための関数呼び出しを示している。ここでは、アフィニティ指定のためのマスクが、計算機におけるすべてのコアのビットマップとなっている(Figure 22)。もしも異なるマスク値を引数に *sched_setaffinity* が再度呼ばれたならば、OS はそのスレッドを指定されたプロセッサに移動する。

```
unsigned long mask = 1; /* processor 0 */

/* bind the calling process to processor 0 */
if (sched_setaffinity(0, sizeof(mask), &mask) <0)
        {
        perror("sched_setaffinity");
        }
```

**Figure 21. Code snippet using bitmap mask for cache affinity scheduling.**

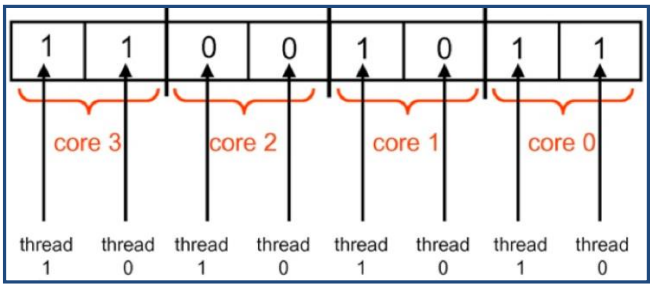



**Figure 22. Affinity mask.**

## 4.20 Event Based Parallel Programs

不規則な方法で通信し合う半ば独立したタスクのグループにアプリケーションが分解される場合や、タスク間通信がタスク間データフロー（タスク間の順序制約）によって決定される場合には、イベントベースの協調手法を使用するのがよい。これらのタスクは、通信も含め、並行して実行できるように実装可能である。この方法は、線形構造でなければならないといった制約や、データフローが単一方向でなければならないといった制約はないが、通信が不規則、時には予測不可能な間隔で起こる可能性がある。

コインランドリーを例に説明する。コインランドリーには利用可能な洗濯機と乾燥機がたくさんあり、人がランダムなタイミングでコインランドリーを訪れる。洗濯機が空いているならば係員によって利用できる洗濯機に案内され、もし空いていなければ順番待ちをする。洗濯機は一度に一回分の洗濯物しか洗濯できない。このとき、与えられた到着時間の分布に対し、コインランドリーでの平均利用時間（洗濯時間と待ち時間の合計）と待ち行列の平均長を計算したい。このシステムにおける「イベント」には、係員まで到達した洗濯物、洗濯機に案内された洗濯物、そして洗濯が終わった洗濯物がある。モデルを簡単化するために、コ

インランドリーに到着する、または離れる洗濯物を、「ソース」オブジェクト、「シンク」オブジェクトとして考える。また、洗濯が終わると必ず係員に知らされ、係員は洗濯機が空いているかどうか分かっているものとする。

これをプログラミングの観点から見ると、アプリケーション開発者は、逐次的な処理として記述される従来のプログラミングではなく、イベントベースプログラミングの中で一群のイベントハンドラを記述するのがよい。入力イベントがこれらのイベントハンドラを起動する。相互作用するプロセスの数が増加した場合、多種多様なイベントへの応答が連続してコーディングされているプログラムを理解することは難しい。いつでも、一つのプロセスは他の多くのプロセスに作用する可能性があり、これらの作用の一つ一つはプロセス自身の状態情報を必要とする可能性がある。このことは、相互作用するプロセスを扱うための抽象概念の必要性を意味している。

## 4.21 Implementing Loop Parallelism

ループ並列（データ並列）は並列性を実装する上で潜在的に最も簡単な手法であり、そのうえ最高の速度向上およびスケーラビリティを達成可能である(Figure 23)。ループをプロセッサ数に分割することによって、それぞれのスレッドの作業量は等しくなる。ループ反復間に依存性はなく、ループ回転数も十分大きいならば、十分なスケーラビリティを達成できる。加えて、一回のループがおおよそ同じ時間となり、プログラムが負荷均衡問題から解放される傾向がある。

```
#include <pthread.h>

struct ThreadParam {
        int startIndex_;
        int endIndex_;
        int threadNb_;  // thread number
        pthread_t threadID_; // pthreads thread object
        VectorsStruct *vectors_; // vectors we're going to calculate
};

static void *threadFunction(void *paramPtr) {
   ThreadParam *param = (ThreadParam *)paramPtr; // get our "instructions"
   std::cout<<"Thread: "<<param->threadNb_<<" from "<<param->startIndex_<<" to "
               << param->endIndex_<< std::endl;
   compute(param->vectors,param->startIndex_, param->endIndex_); // compute the vectors
   return NULL;
}

main{}
{
  int nthreads = 8; // Adjusted for the witdth of the machine
  ThreadsParam* paramPtr[nthreads]; // each thread needs it's set of "instructions"
  VectorsStruct vectors(A, B, C); // need something to calculate

  /* Creation of the threads. */
  for(int i = 0; i < nthreads; ++i)
  {
        int startIndex = (size * i) / nthreads; // each thread gets its own start index
        int endIndex = (size * (i + 1)) / nthreads; // each thread gets its own end index
        ThreadParam *paramPtr[i] = new ThreadParam(startIndex, endIndex, i + 1, vectors);
        int status = pthread_create(&paramPtr->threadID_, NULL, threadFunction, paramPtr);
        // check status
  }

  // join on the threads and delete ThreadsParam array.
}
```

**Figure 23. An example of loop parallelism using Pthreads.**

## 4.22 Aligning Computation and Locality

局所性は性能指向コードにおいて重要な検討事項の一つである。メモリアクセス遅延がプロセッササイクル時間と比較して大きい時、アプリケーション性能は低下する。これを避けるには、計算を実行するプロセッサの近くにデータを置いておくのがよい。さらに、マイクロプロセッサのキャッシュがメモリを最も有効に使用できるようプログラムを構成する必要がある。局所性には 2 種類あり、1) 時間的局所性（使用されたメモリ番地は再び使用されることが多い）、2) 空間的局所性（あるデータが参照された後、その付近のデータも使用されることが多い）である。

コードの局所性最適化には多くの方法がある。例えば、キャッシュブロック化は、マイクロプロセッサのキャッシュに合うように、データオブジェクトを再構成する。ループタイリングは、ループ反復をより小さなブロックに分割し、関連データがキャッシュ内に残るようにするための技術でもある。

### 4.22.1 NUMA Considerations

非均一メモリアクセス(NUMA)アーキテクチャは現代のマイクロプロセッサにおいてよく使われるアーキテクチャの一つである。NUMA アーキテクチャでは、全体のメモリ空間は共有されるが、それぞれのプロセッサは、専用のメモリコントローラを有するノードに接続している（Figure 24）。これによりシステム設計者がより大きなシステムを構築できるようになるが、ノード外へのデータアクセスはノード内データアクセスより時間がかかるため、プログラマはデータとメモリの近さに対してより注意深くなる必要がある。メモリへのアクセス時間が一定ではないため、性能を保証するために個別のコード最適化が必要となる。

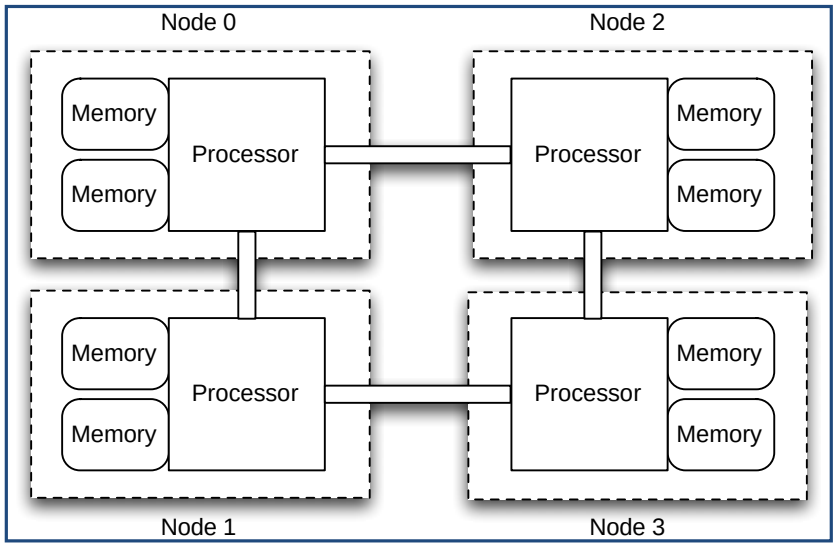



**Figure 24. With NUMA architecture, each processor core connects to a node with dedicated memory.**

### 4.22.2 First-Touch Placement

ファーストタッチ配置では、メモリに最初にアクセスする（書き込む）プロセッサを含む NUMA アーキテクチャ上のノードにそのメモリを割り当てる。ファーストタッチメモリ配置手法は、ほとんどのデータアクセスが各プロセッサに局所化されたメモリに対して行われ

るようなアプリケーションの性能を大いに向上させることができる。ファーストタッチ配置はほとんどの Unix ライクな OS 上で初期設定となっており、多くの場合、配置ポリシを操作する API をサポートしている。

プログラマは、計算を実行するスレッドを使用してデータ構造の初期化を行うよう注意しなければならない。これにより、スレッド実行時にメモリが最も近いノードに配置されることを保証する。さらに、最高のメモリ性能を保証するため、スレッドを、関連するメモリが配置されているノード上のプロセッサに置くべきである。

## 4.23 Message Passing Implementations

MCA はマルチコアシステム内の CPU 間のプロセス間通信(IPC)に関する二つの仕様を作成した。マルチコア通信 API(Multicore Communications API (MCAPI®))とマルチコアリソース（資源）管理 API(Multicore Resource Management API (MRAPI®))である。

MCAPI をシステムに適合させる方法を理解するため、利用する（マルチ）OS と（マルチ）CPU において、システムとリソースを分割する方法を見てみよう。システム自身はハードウェアとソフトウェアを念頭において設計されている。システム要件は何だろうか？単位データあたりの処理に必要な最小遅延は？付帯する決定性要件はあるだろうか？アプリケーションは並列化できるだろうか？

AMP と SMP 間の違いに注意することは重要である。いくつかの状況において SMP 対応のハードウェアは、すべてのコアにまたがる単一の OS を実行させることにより最も効率よく使える。しかしながら、SMP システムのコア間で複数の OS を(AMP システムと同様に)分散させることが、しばしば有利になる。また「ベアメタル」アプリケーションとして知られている、CPU 上で OS 無しにアプリケーションを実行する方法が最適のこともある。

AMP が有利、もしくはシステム設計において必要であると決まると、それらのインスタンス（訳者注：AMP 上で動く個々の OS 上システム）間でのメッセージパッシングのためにプロセッサ間通信(IPC)が必要となる。MCAPI は、特にシステム上のノード間メッセージパッシングを扱う。MCAPI は、OS 間でメッセージを通すためにとても低い層で使うことができ、アプリケーションから見たハードウェアの差異と独自性を抽象化する。

### 4.23.1 MCAPI

MCAPI には、理解すべき 3 つの主たる概念(ノード、エンドポイント、チャネル)がある。MCAPI はノード間メッセージ送受信の実装である。ここでノードとは、CPU、OS インスタンス、スレッド、あるいはプロセスでありうる。初期化ソフトウェアは各ノードを定義し、

ノード間通信を行うための一群のエンドポイントを作成する。このときシステム設計者は適切に構成を決める必要がある。システムは異なる種類のノードを有することができる。たとえば、二つの OS が個々の CPU 上で並存しているとする。第一の OS では CPU 上で実行する 5 つのプロセスとして扱い、第二の OS では OS 全体を単一ノードとして扱う、といったことができる。

ノード間通信には二つのタイプ(コネクションレス型とコネクテッド型)がある。コネクションレス型のエンドポイントはいつでも多くの異なるノードからの通信を受け取ることができる。2 つの異なるエンドポイント間で通信する際にコネクテッド型のチャネルを用いる。この場合、他のノードはそのエンドポイントに接続することはできない。

メッセージパッシングにおいて、チャネルの形態の一つは「ソケットライク」であり、すべてのメッセージが送信順に現れる。チャネルは「スカラーベース」にもなり得て、その場合、単純な整数メッセージがエンドポイント間で渡される。例えばビデオフレームはその全部がデータグラムとして渡されるが、一方でコマンドはスカラーとして渡すことができる。

OS 間でメッセージを渡すためにとても低い層で MCAPI を使うことにより、アプリケーションから見たハードウェアの差異や独自性のすべてを抽象化することができる。付録 D の MCAPI プログラミングのソースコード例を見てみよう。この例は 2 つの OS (Android と Nucleus)で共有するプリンタ機能に関するものであり、初期化とメッセージ処理のためのフローチャートを含んでいる。

### 4.23.2 MRAPI

MRAPI は、（複数の）処理ノードや OS インスタンス間にある、共有メモリや同期オブジェクトのようなリソース（資源）を扱う。API を共有するメモリは、単一ブロックのメモリであり、一つの OS によって所有され、システム内のいくつかのノードによって共有される。これらの API は、ただ一つの書き手と複数の読み手を実現し、それぞれいつでもメモリブロックにアクセス可能である。

同期 API により、異なるノードで実行し、同期可能なアプリケーションを実現できる。MRAPI セマフォはシステム内の任意のノードによってアクセスできるシステム全体のセマフォであり、複数ノードが同時に共有資源にアクセスすることを制限する。

### 4.23.3 MCAPI and MRAPI in Multicore Systems

あるマルチコアシステムにおいて、MCAPI/MRAPI を用いてノード間相互通信フレームワークを提供する方法をみてみよう。この例は 4 コアからなり、それぞれが主記憶から一定量のメモリを割り当てられているとする。また、すべてのコアによって共有されるメモリ領域があるものとする。

まず、MRAPI はシステムノード間でアクセス可能なすべての共有メモリ領域を初期化する。次に MCAPI の初期化中に、エンドポイントやノード間に定義されたチャネルのそれぞれに対して MRAPI 共有メモリが分散して割付けられる。これらのノードはローカルあるいはリモートでありうるが、ノード間メッセージを通すために共有メモリ領域が使われる必要がある。割り込みは、高優先度通信に対するシステム遅延を最小化するため、現在のノード処理を中断する目的で使われる場合がある。

MCAPI と MRAPI の組合せは、いくつかのシステムノードにわたって構成された AMP システムにおいて、フル機能の API を実装するための説得力のある方法である。この通信システムは、他のすべてのノードと通信可能なノードを含むこともできるが、経路を設定することで中間ノードを通過して目的地にデータを届けることもできる。システム設計によって、システム全体で効率的にメッセージを通すことも、通さないこともできる。

### 4.23.4 Playing Card Recognition and Sorting Example

シャッフルされたカードの山を取り、カードを順番に並べ替える、簡単なアプリケーションの例を考えよう。この例はいくつかのタスク(Figure 25)に分割できる。第一のタスクはとても簡単な判別、そのカードは赤か黒か、をする。第二のタスクもまたとても簡単で、赤カードはダイヤかハートか、あるいは黒カードはクラブかスペードかを判別する。このことは、先が尖っているかどうかで決定できる。ダイヤやスペードは尖っており、ハートとクラブは尖っていない。第三は、すべて同じタスクの集合である。割り当てられたカードのマークが何であっても、受け取ってカード順(2-10、ジャック、クイーン、キング、エース)に並べる。

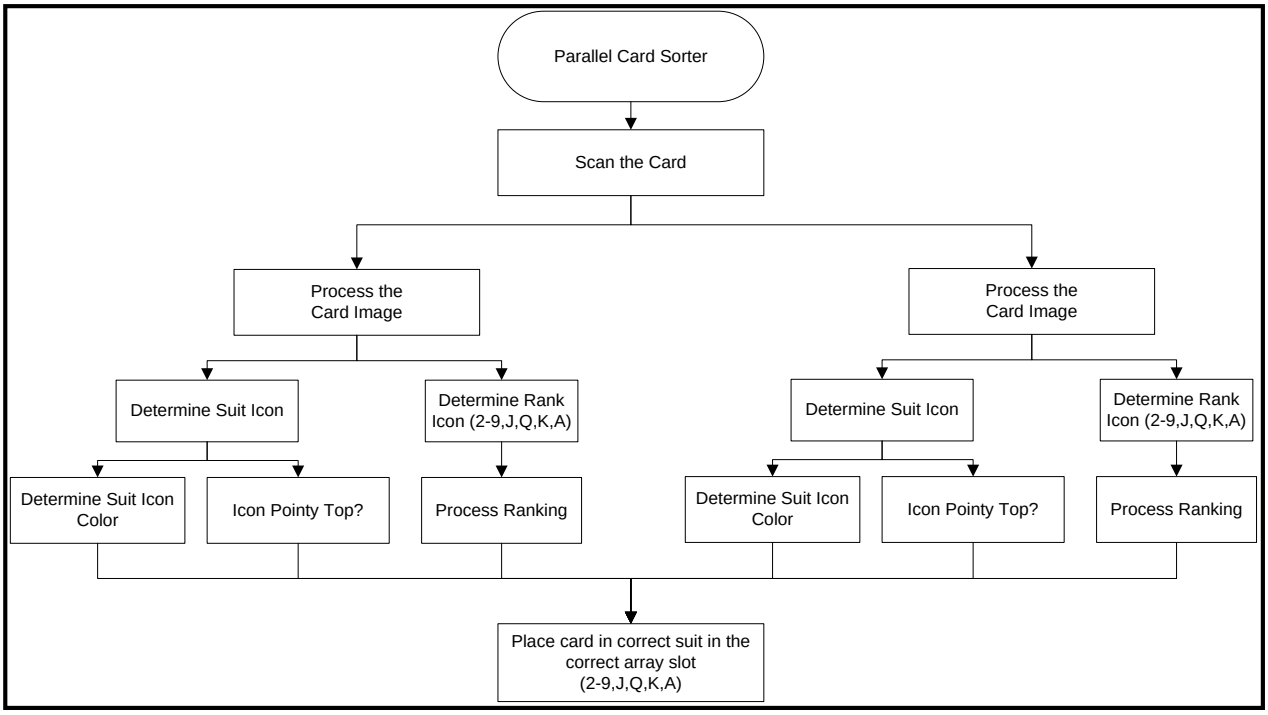



**Figure 25. Playing card recognition and sorting flow.**

今度は、各カードのスーツ（スペード、ハート、クラブ、ダイヤ）とランク（2-10、ジャック、クイーン、キング、エース）を解読するために画像処理を加えることによって問題を複雑にしよう。各タスクはカードをビットマップとして処理する必要があるが、各カードの同じビットにアクセスする必要がある。

エッジ検出アルゴリズムにより、どこに各要素（スーツやランク）が存在するか探索し、ランクの場所とスーツの場所を認識する。今回の場合、もともとのデータセットを保持し、そのデータのコピーを各タスクに渡し、探索や認識を行うため、各タスクを並列実行できる。性質(色、スーツ、ランク)がすべて決定されると、正しい位置にカード画像を配置する。

さて、複数の山を処理することにより、問題をさらに複雑にしてみよう。そこでは、確実に次の山に処理を進めるために同期をとる必要があるだろう。また、このアプリケーションではカードが不足している状況も扱い、一部のデータは時間がかかり過ぎて処理できないかもしれない。

第 3 のタスク（集合）を調べてみる。そこには 4 つの同一な実行スレッドがある。これらのスレッドは、前のタスクによって決定された異なるデータセット上で実行される。カードのランクを決定し、それを順番に並べる。各タスクにおいて、ランクの種類は同じであるが、スーツの種類が違う。そのため、アルゴリズムは同一で、高い並列性がある。これらを 4 つの別々のノードに分配することと、複数スレッドを単一ノードで実行することの、どちらが

より有効であろうか？これは不自然な例かもしれないが、達成すべきことに対し、最適にシステムを分解する方法について理解するのに役立つはずである。

デジタル信号処理プロセッサ(DSP)を用い、最初にカード画像を処理して要素にし、ランクとスーツの場所を認識させよう。その後、このデータは第一のスレッドに渡せる。赤だろうか黒だろうか?　次に別のノードに渡す。上部が尖っているかどうか?　そしてランク画像を第三のノードに渡す。第三のノードは処理により多くの時間がかかるだろうから、複数ノード上で実行する。目的は、カードの特徴を完全に認識し順番に並べて、できる限り速く第三のタスクを完了するアルゴリズムを設計することであったことを、忘れないようにしよう。

### 4.23.5 Using a Hybrid Approach

ノード内で標準 POSIX を使い、ノード間で MCAPI /MRAPI を使うと、さまざまなシステム間で高い移植性を可能にするシステムアーキテクチャを開発することが可能である。しかしながら、負荷を均衡させようと (実行スレッドを他ノードに移動しようと) して、かつてローカルであったものが今はリモートになったり、その逆が起こったりすると、API が変わるだろう。解決策はノード間と同様にスレッド間のすべてのメッセージに MCAPI/MRAPI を使うことである。通信が同じノード内に閉じるならば、MCAPI/MRAPI 層は POSIX のようなノード内メッセージパッシングに変換するだろう。単一ノード上で MCAPI/MRAPI を使うことは、新たな抽象化層を追加することになるが、ノード間/ノード内のすべての通信のための共通の API として MCAPI/MRAPI を使うことで、ノード間負荷分散を可能にするという利点が加わる。通信層として MCAPI/MRAPI を使うことは将来的にシステムアーキテクチャの柔軟性を可能にするが、もし CPU の効率性を最大化する必要があるならば、速さの必要性がその利点より優先されるだろう。

開発中は、単一ノードでも複数ノードでも、すべてのエンドポイント、すべてのタスクにおいて、単一 OS インスタンス上で MCAPI/MRAPI を使うことができ、アルゴリズムを動作できるだろう。これは、マルチコアハードウェアを使う前にシングルコア OS ノード上でアプリケーションをテストする簡単な方法である。すべての並列スレッド動作をテストできるわけではないが、アルゴリズムの論理的性質についてはテスト可能である。

# CHAPTER 5: DEBUG

## 5.1 Introduction

マルチコアシステムを採用するには、並列プログラムを効果的に設計、記述およびデバッグする方法についてプログラマがよく理解している必要がある。並列性は複雑さをもたらし、潜在的に非決定的で非同期なタスクがシステムスケジューラによって割り込まれることで、複雑性はさらに悪化し、デバッグを困難にする。並行性に関連したバグは毎回のアプリケーションの実行に現れないことがあり、同じアプリケーションを連続して実行しても異なる結果が生じる可能性がある。

## 5.2 Parallel Processing Bugs

マルチコアプラットフォーム上で並列プログラミングを行う開発者は、並列およびマルチスレッドのソフトウェア設計開発に熟練していることに加え、アプリケーションの安定性と性能向上を確保するため、利用可能なリソース（資源）に関してよく理解し、調整できることが必要である。

マルチスレッド並列アプリケーションでは、スレッドは任意の時間に割り込まれる可能性があり、潜在的に、可能性のあるすべての命令がすべての実行スレッドを割り込むことになる。マルチスレッドアプリケーションにおいて発生する問題を以下に示す[xxxviii]：

1. あるスレッドが共有資源をロックしたまま、解放前に割り込まれて別の作業に移行すると、別のスレッドがストールすることがある。ロックされた共有資源を必要とする他のスレッドは無限の長い時間待たされる可能性がある。
2. 複数のスレッドが一つのロック取得を待つスレッドコンボイによりスレッドはストールする。すべてのスレッドがロック解放を待つため、システムはすべてのブロックされている実行可能スレッドを切り替えるため、大きなオーバーヘッドとなる。
3. データ競合（Figure 26）：同時アクセスを防ぐための明示的な仕組みがない競合アクセスを実行する、２つの並行スレッドにより起こる。同期やクリティカルセクションを使用してデータ競合を修正する。アプリケーション性能に悪影響を与えないよう、不要な同期は避けるべきである。
4. 階層的ロックによりデッドロックが発生することがある。各スレッドが別のスレッドの動作（ロック解除）を待つことで、すべてのスレッドがブロックされる。
5. ライブロック（Figure 27）：複数スレッドがデッドロックを検出、復帰しようとして発生する。プロセス/スレッドが互いに検出、復帰を繰り返し続け、デッドロック検出アルゴリズムを起動することによって起こる。

訳者注：Figure 26 と 27 は 28 と同じである。現在 MCA にて差し替え検討中。

```
char **image;
int size_x, size_y;

void *CreateThreads(void *arg)
{
  pthread_t threads[NUM_THREADS];
  int i = 0;
  for (i = 0; i < NUM_THREADS; ++i) {
    if (pthread_create(&(threads[i]), NULL, FilterImage,
    (void*)i)){
      ErrorMessage("pthread_create failed!\n");
    }
  }
  for (i = 0; i < NUM_THREADS; ++i) {
    if (pthread_join(threads[i], NULL)) {
      ErrorMessage("pthread_join failed!\n");
    }
  }
  CleanExit(0);
  return NULL;
}

void* FilterImage(void *id)
{
  int cpu_num = (int)id;
  int start_y, end_y;
  /* Assume size_y % NUM_THREADS = 0 */
  start_y = (size_y / NUM_THREADS) * cpu_num;
  end_y = (start_y + (size_y / NUM_THREADS)) - 1;
  /* Filter Image */

}
```

**Figure 26. Code sample show in data race condition.**

```
char **image;
int size_x, size_y;

void *CreateThreads(void *arg)
{
  pthread_t threads[NUM_THREADS];
  int i = 0;
  for (i = 0; i < NUM_THREADS; ++i) {
    if (pthread_create(&(threads[i]), NULL, FilterImage,
    (void*)i)){
      ErrorMessage("pthread_create failed!\n");
    }
  }
  for (i = 0; i < NUM_THREADS; ++i) {
    if (pthread_join(threads[i], NULL)) {
      ErrorMessage("pthread_join failed!\n");
    }
  }
  CleanExit(0);
  return NULL;
}

void* FilterImage(void *id)
{
  int cpu_num = (int)id;
  int start_y, end_y;
  /* Assume size_y % NUM_THREADS = 0 */
  start_y = (size_y / NUM_THREADS) * cpu_num;
  end_y = (start_y + (size_y / NUM_THREADS)) - 1;
  /* Filter Image */

}
```

**Figure 27. Code sample showing a Livelock condition.**

## 5.3 Debug Tool Support

マルチスレッドコードのデバックに対応するツールは、最低限、デバッグ中のスレッド切替とスレッド状態の検査に対応すべきである。非決定的なスレッドスケジューリングやプリエンプション、そして制御フローやデータの依存関係によって、マルチスレッドアプリケーションはより複雑になる[xxxix]。ランタイムスレッドスケジューリングやコンテキストスイッチなどによる、複数命令列の非決定的な実行は、本質的に OS スケジューラによって引き起こされている。そのため、スレッドの相互作用に起因するデッドロックや競合状態などの問題は、デバッガによって隠蔽される可能性がある。スレッド優先度、プロセッサアフィニティ、スレッド実行状態、そしてスタベーション時間などの要因がリソースやスレッド実行に影響を与える可能性がある。

非決定性のためスレッド関係のバグは発見するのが困難であり、特定のバグが特定の順番でスレッドが実行されたときのみ発見される場合がある。よく見られるバグには、データ競合、デッドロック、そしてスレッドストールがある。同期によってデータ競合が解決するかもしれないが、スレッドストールやデッドロックを引き起こすかもしれない。

並行システムのデバッグには多くの手法があるが、その中には伝統的デバッグやイベントベースデバッグが含まれる。伝統的デバッグ（ブレークポイントデバッグ）は、並列プロセスごとに 1 つの逐次用デバッガを使用することにより、並列プログラムに適用されてきた。このようなデバッガは、プロセス間に相互作用があると限られた情報しか得ることができない。イベントベース（またはモニタリング）デバッガはマルチスレッドアプリケーションの機能をある程度再現することができるが、オーバーヘッドが大きくなってしまう。デバッガは、データのテキスト表示やプロセス時間の図示、またはプログラム実行のアニメーションのような手法によって制御フローを表示する[xl]。

使用するスレッド API はデバッガの選択に影響を与える可能性がある。例えば、OpenMP を使用する場合は OpenMP 対応のデバッガを使用する必要があり、これによってスレッドコード生成後に OpenMP の構文に従った構成要素や変数（private、shared、thread private）の型のような情報へアクセスすることができる。

一般的に逐次最適化の使用は、デバッグ情報の品質に悪影響を与える可能性がある。デバッグ情報の品質向上のためには、高度な最適化を解除するのが望ましい場合がある。逐次最適化によるデバッグへの影響の例を以下に示す[xli]：

- インラインソースコードをステップ実行すると、異なる場所が表示されて混乱する可能性がある。
- コード移動（コードをループ外に移動する）は異なる場所にコードが散在する可能性がある。

- x86アプリケーション最適化におけるベースポインタの使用は、誤ったバックトラックを引き起こす可能性がある。

複雑さとデバッグの諸問題を最小化するために、並列アプリケーションを記述するプログラマは、並行性に関連するバグをできるだけ多く防ぐコードを記述しなければならない。並行性に関連するバグを最小化するための推奨として、可能ならばログを有効にする、アプリケーションが使用するタスク数をパラメタ化した並列化により、並列化レベルを制御できるようにする、といったことがあげられる。

## 5.4 Static Code Analysis

静的コード解析は、アプリケーション実行や特別な入力考慮の必要がなく、計測化コードやテストケース開発を要求することもなく、ソースコードに対して行われるものである。静的コード解析は、デッドロックやライブロックがないことなどの性質の正確性を保証するために、すべてのデータ範囲を含むすべての実行可能パスを網羅的に探索する。静的コード解析ツールは絶対的な時間（実際の実行時間）をモデル化することはできないが、相対的な時間や時間的順序をモデル化することはできる。静的コード解析の一つの手法は、プログラムの構文木から生成された有向制御フローグラフを使用する方法である。変数に関連する制約は木構造のノードに割り当てられる。ノードはプログラム上のある地点を表し、制御の流れはエッジで表される。制御フローグラフに基づく分析を使用して検出される典型的なエラーを以下に示す：

- 不正な引数の数、型
- 終了しないループ
- 到達不可能なコード
- 初期化されていない変数、ポインタ
- 範囲外の配列要素
- 変数または構造体ポインタへの不正アクセス

静的デッドロック検出ツールは、プログラミング慣習からの違反を探す。型システムやデータフロー解析、モデル検査といった手法に基づいて作られてきた。型システムは、計算される値の型により、「フレーズ」を分類する構文フレームワークである。型検査は、デッドロック検査のために、プログラムが正しく型付けされているか評価する。これらのシステムには注釈が必要であり、大規模なシステムには向いていない。

データフロー解析では、コールグラフ解析を使用してデッドロックのパターンを検査する。静的なロック順序グラフを生成し、デッドロックする可能性があるループを報告する。この

解析では型情報や定数値、null かどうかなどの情報を提供する。データフローアルゴリズムは一般的に条件テストからの情報を使用する。

モデル検査はすべてのプログラムの振る舞いを系統的に検査する。また、入力プログラムが有限で扱いきれる状態空間を有すると仮定している。潜在的なエラーを検出するため、ソースコードを使用してソフトウェアアプリケーションをモデル化する。これらのモデルによって振る舞いの特徴を解析することができる。マルチスレッドアプリケーションでは膨大な数のスレッド入替が発生するため、モデル検査は計算コストが高く、大規模なプログラムに適していないため、適用範囲が限られる[xlii] [xliii] [xliv]。

静的コード解析を使用することでコードの品質を維持することができる。ビルドプロセスに静的コード解析を統合することは、開発プロセスの初期段階で潜在的な問題を見つけるのに役立つため、推奨される。静的コード解析ツールを使用する場合は以下に気を付ける：

1. 可能であれば冗長(verbose)オプションを使用する。複数の問題に対する多くの例を用いてソースコードを解析することは圧倒的な効果がある。より重要な問題を検出できるように新たな検査の段階的導入を検討することが望ましい。
2. 誤検知に注意する。静的解析では１００％の精度を保証できないため、ソースコードの変更を行う前に報告された問題を確認することが望ましい。

静的コード解析は開発・デバッグプロセスを補完するが、確立した開発・テストプロセスを置き換えるということではなく、ほかのデバッグ手法と組み合わせて使用するべきである。

## 5.5 Dynamic Code Analysis

動的コード解析は、物理ハードウェアまたは仮想プロセッサ上に構築されたプログラムを実行して行う。計測化コードやメモリアクセス解析を使用することで問題をより正確に検出できる。動的コード解析ツールは一般に誤検知率が低く、容易に自動化できる。動的テストを効率的にするために、適切なコードカバレッジを実行するテストケースを選択する必要がある。

マルチスレッドの関連で言うと、動的スレッド解析ツールを使用してスレッド関連のバグを見つける場合、以下の推奨事項が役立つ[xlv]：

1. ベンチマークでは、アプリケーションのうちスレッド化された部分を実行すべきである。スレッド化されたコードを実行しない限り、ツールはスレッド関連の問題を特定、解析することができない。

2. 使用したテストスイートにより、動的解析ツールが鍵となるコード領域を適切に実行したことを確かめるため、コードカバレッジツールを使用する。
3. シンボルや行番号情報を利用するため、最適化されていないデバッグ版のアプリケーションを使用する。
4. アプリケーションに計測化コードを追加する場合には、限定した方がよい。もしスレッドがアプリケーションの特定の領域に限定されている場合は、アプリケーション全体を計測する必要はないかもしれない。
5. 可能であれば、最適化版と非最適化版の両方のアプリケーションをテストする。特定のバグが最適化版アプリケーションにのみ検出される場合は、選択した最適化が疑わしいバグに寄与していないかを確認するべきである。
6. バイナリ計測化を使用する場合は、バイナリは再配置可能にしなければならない。

以下に、動的コード解析ツールを使用するためのいくつかのヒントを示す：

- ワーキングセットのサイズを制限する。
- ツールの出力で頻繁に観測される変数から解析を始める。
- 誤検知に注意する。

Valgrind に基づくオープンソースの動的スレッド解析ツールには ThreadSanitizer や Helgrind、DRD などがある[xlvi]。

## 5.6 Active Testing

アクティブテストは 2 フェーズから成る。最初のフェーズでは、静的コード解析や動的コード解析を用いて、アトミック性違反やデータ競合、デッドロックなどの並行性に関わる問題の可能性を検出する。この情報は次のフェーズにおいてスケジューラの入力として与えられ、これらの並行性問題を（スケジューラを制御することによって）実現し、検出する際の誤検出を最小化する。このアプローチを用いているツールの一つとして CalFuzzer がある[xlvii]。

## 5.7 Software Debug Process

組込みマルチプロセスやマルチスレッドアプリケーションのソフトウェアデバッグは難しい問題であり、最小限の労力で最大限のバグ検出ができるように、ソフトウェア開発プロセスを綿密に考えることが要求される。以下の手順は、初期の逐次版アプリケーションから並列タスクを含むものへと進化させ、ターゲットマルチコアプロセッサ上で実行させる、段階的なデバッグプロセスである：

1. 逐次版アプリケーションをデバッグする。
2. （バグに対する）防衛的 (defensive) コーディング手法により、逐次版を並列化する。
3. 並列版を逐次的に実行させデバッグする。
4. 並列タスク数を増やしながら並列版をデバッグする。

### 5.7.1 Debug a Serial Version of the Application

マルチコアプロセッサを利用するアプリケーションでは、バグは 2 つのカテゴリに分けることができる。1)並列処理とは関係ない一般的なバグと 2)並列処理のバグである。並列アプリケーションのデバッグは、逐次アプリケーションのデバッグより本質的に難しいため、逐次版アプリケーションからデバッグを始め、並列実装に関係ない全てのバグを見つけて修正するのがよい。伝統的なデバッグ手法はこれらの問題を見つけるのに十分であるが、その技術の詳細はこの文書の範囲外である。

最終的にヘテロジニアスマルチコアプロセッサ上で動作するアプリケーションの逐次版を作成することは難しいこともあるが、そのような場合、ソフトウェア実行をエミュレートするライブラリを使えることもある。

### 5.7.2 Use Defensive Coding Practices

一般的なバグを最小化することでよく知られているコーディング手法は、並列処理のバグを最小化するのにも役立ち、利用すべきである[xlviii]。マルチコア開発に特有の推奨事項は 3 つあり、1)ログ記録の有効化、2)並列性のパラメタ化、3)同期ポイント追加、である。ログ記録の有効化には、アプリケーション状態を追跡する、時間とタスクを結びつけるようなソースコードを追加する、といったことがあげられる。アプリケーション並列性のパラメタ化には、アプリケーション実行時の並列タスク数を容易に変更できるようにコードを追加する、といったことがあげられる。一般に、この推奨事項は、ここで述べられるデバッグ容易性だけでなく、アプリケーションの性能スケーリングを将来試みる際にも役立つ。同期ポイントについては詳細をこの章内で後述する。

### 5.7.3 Debug Parallel Version While Executing Serially

最初の並列実装完了後、それを最小限の並列性を用いてデバッグする。このステップは、前ステップで述べた並列性のパラメタ化が容易にできるかどうかに依存している。逐次一貫性の話はこの章内で後述する。

このステップは、単一コアプロセッサにおいて GUI をスレッド化して応答性を向上させるような、待ち時間のためにスレッドを用いるアプリケーションには適用できない場合がある。

ヘテロジニアスマルチコアプロセッサで実行するアプリケーションも含めて逐次的に動作させることが不可能な場合もある。代わりに、可能な限り並列実行を制限すること、例えば、プロセッサコアのタイプ毎に並列実行を一つのタスクに制限することは効果的である。

### 5.7.4 Debug Parallel Version Using an Increasing Number of Parallel Tasks

並列版の逐次実行デバッグの後は、順次並列性を増やしていくのがよい。このステップでも、並列性のパラメタ化を前提としている。デュアルコア版をデバッグし、次はクアッドコア版、といったように増やしていく。

## 5.8 Code Writing and Debugging Techniques

本節では、以前の節で述べた、個別推奨事項のいくつかについて詳細を述べる：

- 逐次一貫性
- ログ記録
- 同期ポイント
- スレッド検証
- シミュレーション
- ストレステスト

### 5.8.1 Serial Consistency

逐次一貫性を強制することで、逐次版ソフトウェアから並列版ソフトウェアへの移行を容易にする。逐次一貫性とは、逐次版と並列版が十分に類似していて、簡単に入れ替えることができるようなコードの特性である。次の例(Figure 28)は画像フィルタリングアプリケーションの逐次版、並列版の両方を示したものである。プログラムの逐次版と並列版はそれぞれ以下のコンパイルコマンドによって有効になる：

- *gcc –DNUM_THREADS=1 app.c*
- *gcc –DNUM_THREADS=2 app.c*

並列版(*NUM_THREADS=2*)ではスレッド数は 2 であり、*NUM_THREADS* の定義が、コンパイル時にコマンドラインで設定される。*FilterImage()*関数はスレッド数にもとづいてデータを分割するコードのみを含む。

```
char **image;
int size_x, size_y;

void *CreateThreads(void *arg)
{
 pthread_t threads[NUM_THREADS];
 int i = 0;
 for (i = 0; i < NUM_THREADS; ++i) {
  if (pthread_create(&(threads[i]), NULL, FilterImage,
  (void*)i)){
   ErrorMessage("pthread_create failed!\n");
  }
 }
 for (i = 0; i < NUM_THREADS; ++i) {
  if (pthread_join(threads[i], NULL)) {
   ErrorMessage("pthread_join failed!\n");
  }
 }
 CleanExit(0);
 return NULL;
}
void* FilterImage(void *id)
{
 int cpu_num = (int)id;
 int start_y, end_y;
 /* Assume size_y % NUM_THREADS = 0 */
 start_y = (size_y / NUM_THREADS) * cpu_num;
 end_y = (start_y + (size_y / NUM_THREADS)) - 1;
 /* Filter Image */

}
```

**Figure 28. Serial consistency example.**

### 5.8.2 Logging (Code Instrumentation for Meta Data Send and Receive)

アプリケーションにおける状態やイベント順序を追跡する優れた手法は、実行時にコード内でトレースバッファを用いてログを生成することである。この手法では、実行後に取り出し、問題が起きた場合にアプリケーション実行を追跡できるように、識別子、タイムスタンプ、メッセージを含むログを記録するためのトレースバッファを作成する。トレースバッファを利用する場合は、トレースバッファへのアクセスが、アプリケーション内のどこにあるかに注意しなければならない。トレースバッファへのアクセスを制御するための同期が、並行性のバグを隠してしまう場合がある。

Figure 29 は、LOG_SIZE サイズのログ構造体に対しログを追加する *event_record()*関数を示している。配列要素を指す変数 *id_count* をインクリメントする部分で排他制御していることに注意して欲しい。時間はシステム関数 *time()*によって取得しているが、これはスレッドセーフであると仮定している。より細かい粒度が望ましい場合、クロックサイクル情報が使える可能性がある。

```
#define LOG_SIZE 1024
#define MSG_SIZE 64
typedef struct log_entry_s {
 unsigned int task_id;
 time_t timestamp;
 char message[MSG_SIZE];
} log_entry;
log_entry event_log[LOG_SIZE];

int id_count = -1;

int event_record(char *message) {
 time_t temp_time = time(NULL);
 pthread_mutex_lock (&mut);
 id_count++;
 id_count = id_count % LOG_SIZE;
 pthread_mutex_unlock (&mut);
 event_log[id_count].task_id = (unsigned int)pthread_self();
 event_log[id_count].timestamp = temp_time;
 strncpy(event_log[id_count].message, message, MSG_SIZE);
 event_log[id_count].message[MSG_SIZE-1] = '\0';
 return 1;
}
```

**Figure 29. Logging example.**

### 5.8.3 Synchronization Points

同期ポイントは、Figure 28 に示されるように、他のタスクがある特定の場所に到達するまで、与えられたタスクが待つべきアプリケーション内の特定の場所である。*CreateThreads()*を実行しているスレッドは、*pthread_join()*を利用して *FilterImage()*を実行している他のスレッド

を待っている。同期ポイントとログ記録は、プログラマが実行中のアプリケーションの状態を理解するために、アプリケーション全体で用いるのがよい。注意:プログラムのホットスポットでの同期ポイントの利用は、性能に影響する可能性があるため、避けるべきである。

### 5.8.4 Dynamic Analysis Techniques Summary

動的分析ツールは、アプリケーション実行時に動的に分析する。例えば、マルチスレッドにおいてデータ競合を見つけることを考える場合、ツールは、スレッド実行中のすべてのメモリアクセスを監視する。他のスレッドによりアクセスされるアドレスと比較し、そのアクセスがなんらかの同期によって保護されているかどうかを調べることで、read-write や write-write 競合を検出できる。動的分析は、複数スレッドによる明確な変数アクセスエラーだけでなく、ポインタを介して間接的にアクセスされるメモリのエラーも見つけることができる。

典型的な動的解析ツールには、分析実行直前のバイナリファイルに計測コードを直接挿入するもの(バイナリ計測化)や、コンパイル時に挿入するもの(ソース計測化)がある。計測化や検証に関する技術の詳細は Banerjee et al.の論文を参照されたい[xlix]。

### 5.8.5 Simulation Techniques Summary

シミュレータは最終的なデバイスハードウェアを模倣して、開発中の 2 つのポイントで潜在的な問題を発見するのに役立つソフトウェアツールである。まず、多くの組込みプロジェクトではハードウェアが利用できるようになる前にソフトウェアの開発が必要となるため、ハードウェアプラットフォームの代わりにシミュレータを利用する。早期のシミュレータの使用は、並行アプリケーションのテストとデバッグのためのプラットフォームとして機能する。さらに、シミュレータは容易に設定変更可能であるため、ストレステストのプラットフォームとして利用できる。

### 5.8.6 Stress Testing

先述したように、並行性のバグは非決定的である。したがって、マルチコアプロセッサを利用するアプリケーションは、デュアルコアプロセッサでは正しく動作するがクアッドコアプロセッサでは正しく動作しない場合や、もしくはその逆の場合も起こる可能性がある。ストレステストは、アプリケーション実行に使われるスレッド数やタスク数を変更し、加えてコア数や様々なシステム構成要素の遅延時間などのシステム設定値も変更することで、並行性に関するコード内の問題が見つかる機会を増やす。

ストレステストの効果的な利用法として、上記のような数多くの設定値変更をして、アプリケーションが正しく動作しないケースを見つける方法がある。これは、タイミング依存のために実際のハードウェアでは滅多に起こらない異常を見つけるのに有効である。

# CHAPTER 6: PERFORMANCE

## 6.1 Performance

マルチコアアーキテクチャはコードの作りにとても敏感である。並列実行は、期待される性能向上を阻害する多くのオーバーヘッドの影響を受ける。この章では、主要なボトルネックによる性能影響を回避または軽減するためのソフトウェアアプリケーションのチューニング方法を説明する。ただし、並列 I / O、OS レベル（動的メモリ割当、プロセススケジューリングなど）、低レベル通信（DMA 転送効率など）の最適化は扱わない。なぜなら、これらは通常、個別の組込みシステムに特有なものだからである（ただし、これらのレベルには一般にかなりの改善の余地がある）。2 つ以上の並列プログラミングモデルを混在させたハイブリッド並列プログラミングについても明示的には取り上げないが、これらも性能最適化のためには考慮する必要がある。

後で見るように、マルチコアプロセッサを用いた性能改善は、ソフトウェアのアルゴリズムとその実装に依存する。並列の問題というものは理想的に、はるかに遅い主記憶を避け、各コアのキャッシュ内に問題が収まるように分割されている場合に、コアの数に近いまたはそれ以上にもなる性能向上を実現する可能性がある。しかしながら、多くのアプリケーションは、その開発者が全体をリファクタリングするために多大な労力を費やさない限り、大幅に高速化することはできない。

## 6.2 Amdahl’s Law, Speedup, Efficiency, Scalability

並列システムの性能のは高速化の観点で述べられており、その割合は以下の通りである。

$$S_P = \frac{T_1}{T_P}$$

ここで、$T_1$ は 1 コアでの実行時間で、$T_P$ は $P$ コアでの実行時間である。そして目標は $P$ 倍の速度向上を行うことである。そのため並列プログラムのふるまいを特徴付けるため、効率性($E_P$)を用いる：

$$E_P = \frac{T_1}{P * T_P}$$

効率が 1 になる場合、並列プログラムはハードウェアを最大限に使っているということになる。コア数が増えても効率が 1 に近いままのとき、スケーラブルであるという。実行する作業量を増やすことなくコア数を増やした時にスケーラビリティが達成されることはほとんどない。

Amdahl の法則は、逐次実行部分が並列化の潜在的なメリットに制約をかけることを記述したものである。*P* コアを用いたときの実行時間 $T_P$ は、以下で与えられる。

$$T_P = seq * T_1 + (1 - seq) * \frac{T_1}{P}$$

ここで、*seq*( $\in$*[0,1]*)は、本来的に逐次実行される部分の割合である。たとえば、*seq = 0.5* の場合、並列実行の潜在的な性能向上は高々2 倍である。通常、スケーラビリティは、問題のサイズが大きくなることによって達成される。これは、処理する問題の増大とともに比率 *seq* が減少するためである。

## 6.3 Using Compiler Flags

より高い性能を達成するための並列化を検討する前に、最初に逐次実行を効率的にする必要がある。以下の節では、逐次コードチューニングについて説明する。まず、コンパイラフラグを利用する方法から見ていく。

コンパイラには、コンパイルフラグによって有効化される多くの最適化機能が実装されている。最も一般的で単純なものは-O1、-O2、-O3 であり、ローカルな最適化から始まり、-O3 ではより広範な最適化を行う。これらのフラグは一群の最適化機能を有効化するが、それらはさらに特定のフラグによって設定または設定解除できる。最も積極的な最適化モードは、コンパイル時間とデバッグの労力を大幅に増加させる可能性があるため、通常はデフォルトでは設定されない。

より複雑な指定フラグには、関数間最適化、プロファイルに基づく最適化、自動ベクトル化または並列化を可能にするものや、プログラム解析を支援する（no alias フラグなど）ものがある。いくつかのフラグは、特定の関数内（浮動小数点、関数インライン展開など）を扱う。これらのフラグは、アプリケーションによって性能への影響が大きく異なる可能性があるため、すべてテストする必要がある。

以下は、いくつかの一般的知見である：

1. デバッグフラグ '-g'を設定すると、デバッグ機能を維持するため、最適化を制限するコンパイラもある。あるいはコンパイラの中には、最適化レベルを上げたときにデバッグ情報を減らすものもある。
2. 最適化フラグの適切な組み合わせを見つけることは、手動であるため面倒である。
3. 最高レベルの最適化（-O3など）が常に最速のコードを生成するとは限らない。なぜなら、このモードでは、コンパイラは最適化間のトレードオフを調整する傾向があり、時には誤った選択をすることがあるからである（実行時データがないため）。

4. 最適化によってプログラムの結果が異なる場合がある（例：浮動小数点数の丸め結果の違い）。
5. コンパイラには正しいコードを生成する使命がある。最適化の安全性が証明されない場合、その最適化は破棄される。わずかな書き換え（例えば、ポインタエイリアス除去）の後、コンパイラが最適化を適用できるようになることもある。

## 6.4 Serial Optimizations

逐次最適化においては、データ構造に関する情報を提供し、ベクトル化のための命令レベル並列性とデータレベル並列性をコンパイラが検出しやすくするため、アプリケーションコードを書き換える。この節では、最初にエイリアス情報について述べる。次に、主にループ構造に適用される大規模な再構成手法を紹介し、最後に、SIMD 命令について説明する。

### 6.4.1 Restrict Pointers

多くのコンパイラ最適化は、ポインタ間の潜在的なメモリエイリアスによる偽のデータ依存によって制約を受ける。そのようなポインタは不用意に使うべきではないが、避けられないときは restrict ポインタ（ISO / IEC 9899：1999 標準（いわゆる C99）で導入）を使用し、あるメモリ領域を指すポインタが、そのメモリ領域を指す唯一のポインタであることを宣言する。これはコンパイラの 「エイリアス解析」を支援する。Figure 30 の例では、ポインタの制約指示（“restrict”宣言）がなければ、コンパイラは、パラメタ dest および src が重なり合う（すなわち、エイリアスする）と想定しなければならない。

```
/* File: restrict.c */

void f (int* restrict dest, int * restrict src, int n) {
 int i;
 for (i=0;i < n; i++){
  dest[i]=src[i];
 }
}
```

**Figure 30. Code sample restricting pointers.**

以下は、いくつかの一般的知見である：

1. C ++はまだ'restrict'をサポートしていない。 エイリアス制限は、コンパイラフラグ（例えば、'fno-alias'や 'fargument-noalias'）を使用して指定することもできるが、これらのオプションはアプリケーションソフトウェア内にある依存関係を壊す可能性があるので注意して使用する必要がある。

2. エイリアスが存在するにもかかわらず間違ったrestrict指定をした場合、バグ検出が困難になることがある（特にデバッグ機能有効化が最適化を制限してバグを隠す場合）。
3. いくつかのコンパイラでは、エイリアス知識に欠ける場合、複数種類のコード（エイリアス想定とエイリアス無しのもの）が生成される。これはコードサイズに影響を与える可能性がある。

### 6.4.2 Loop Transformations

ループは、ほとんどのアプリケーションにおいて典型的なホットスポットである。そのため、プロセッサの内部構造に適合するようにループ内の計算とメモリアクセスの順序を並べるため、ループ変換が提案されている。

Figure 31 は、命令レベル並列性（ILP）を明示し、メモリアクセス数を減らす強力な変換である外部ループ展開（アンロールアンドジャム(*unroll-and-jam*)）を示している。外側ループの２回転を内部ループ内で展開する。これにより共通の部分式 *v[j]*をくくりだすことができ、ループ本体内部の独立演算を増やす。ただし、ループ変換は常に安全に適用できるわけではないことに注意すべきである。たとえば、実行順序を制約する依存関係（第 3 章を参照）を保存していることを確かめる必要がある。

```
/* File: unrollandjam.c */

//original code
void fori (int dest[100][100], int src[100][100], int v[]){
 int i,j;
 for (i=0;i < 100; i++){
  for (j=0;j < 100;j++){
   dest[i][j] = src[i][j] * v[j];
  }
 }
}
// after unroll-and-jam
void ftrans (int dest[100][100], int src[100][100], int v[]){
 int i,j;
 for (i=0;i < 100; i = i+2){
  for (j=0;j < 100;j++){
   int t = v[j];
   dest[i+0][j] = src[i+0][j] * t;
   dest[i+1][j] = src[i+1][j] * t;
  }
 }
}
```

**Figure 31. Sample code showing the unroll-and-jam loop transformation.**

以下は、いくつかの一般的知見である：

1. 一部の変換は、コンパイラ固有のプラグマを使用して実装できる（例：*#pragma unroll(2)*）。
2. ループ変換の正当性を手作業で確認することが必要であるが、必ずしも自明ではない。
3. 手動でループ変換を実行すると、ターゲット依存コードになり、読みにくく保守を困難にする可能性がある。

### 6.4.3 SIMD Instructions, Vectorization

SSE や AltiVec などの SIMD 命令は、レジスタ内に詰め込んだベクトル化データを並列処理することによって性能向上させる強力な方法である。例えば、128 ビットレジスタは、64 ビット 2 ワードベクトルまたは 32 ビット 4 ワードベクトルとして見ることができる。

SIMD 命令を活用する方法として、インラインアセンブリコード記載、組込み C ライブラリ利用、コンパイラによる自動ベクトル化、がある。

Figure 32 は、SSE 組込みライブラリを使用したサンプルコードを示している（‘m128d’は 2 つの倍精度浮動小数点数を表すベクトル型）。SSE 命令は、組込み関数 *mm_load_pd*、*mm_mul_pd*、*mm_add_pd*、*mm_store_pd* を介して使用される。SSE 版は、反復ごとに 1 つではなく 2 つの結果を生成する。

```
/* File: simd.c */

//original code
void daxpy( unsigned int N, double alpha, double *X, double *Y ) {
 int i;
 for( i = 0 ; i < N ; i++ ) {
  Y[i] = alpha*Y[i] + X[i];
 }
}
//SSE code
void daxpySSE( unsigned int N, double alpha, double *X, double *Y) {
// m128d is two 64 bit float.
m128d alphav = _mm_set1_pd(alpha);
int i;
for( i = 0 ; i < N ; i+=2 ) {
  __m128d Yv   = _mm_load_pd(&Y[i]);
  __m128d Xv   = _mm_load_pd(&X[i]);
  __m128d mulv  = _mm_mul_pd(alphav, Xv);
  __m128d computv = _mm_add_pd(mulv, Yv);
  _mm_store_pd(&Y[i], computv);
 }
}
```

**Figure 32. Example code using the SSE intrinsic library.**

以下は、いくつかの一般的知見である：

1. 一部のプロセッサには、SIMDメモリアクセス命令にデータアラインメント要件があり、コンパイラプラグマを使用して指定できる。
2. 最も単純なケースでは、コンパイラはSIMD命令を使用したベクトルコードを自動生成する。
3. 手書きのアセンブリコードを保守することは困難である。高級言語での組込み関数の使用が望ましい。
4. 飽和演算、述語付き(predicated)演算、およびマスク演算は、使用可能な場合、性能向上に極めて有用である。

## 6.5 Adapting Parallel Computation Granularity

第 3 章で説明したように、並列実行は常に、タスク起動やタスク間同期、データ通信、（メモリ整合性のような）ハードウェア記憶処理、（ライブラリ、ツール、ランタイムシステムなどの）ソフトウェアオーバヘッド、およびタスク終了のような関数から生じるオーバーヘッドを招く。

一般論として、小さい（細粒度）タスクは非効率である。ほとんどの実装において、すべてのコアを動作状態に保つために十分多くのタスクを持つことと、オーバーヘッドを見えなくするために各タスクの計算量を十分大きくすることとの間には、トレードオフの関係がある。例えば、Figure 33 は、ソート、計算幾何学、FFT のようなアルゴリズムに使われる分割統治法の計算構造を示している。*divideAndConquer* 関数は再帰的にタスク(*parallel_combine*)を生成し、十分に小さい問題（*basecase*）に達すると新しいタスクは生成されなくなる。粒度は *basecase* のサイズを固定することによって調整される。

```
/* File: granularity.c */

void divideAndConquer (int *p){
 pthread_t t1,t2;
 if (basecase(p)){
   p[3] = basesolve(p);
 } else {
  int p1[3],p2[3];
  get_part1(p1,p);
  pthread_create(&t1,NULL,(void *(*)(void *)) divideAndConquer,p1);
  get_part2(p2,p);
  pthread_create(&t2,NULL,(void *(*)(void *)) divideAndConquer,p2);
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
  p[3] = parallel_combine(p1,p2);
 }
}
```

**Figure 33. Divide and Conquer approach.**

多重ループを処理する場合、粒度を調整するために多くの手法が使用可能である。これらの手法は概念的には理解しやすいが、正しく実装することは、実際には複雑である。

以下の変換を利用して、タスクの計算量を増やすことができる：

- ループ結合（ループフュージョン (loop fusion)）（またはループマージ(loop merging)）とよばれる変換では、独立した計算を行う連続した複数のループを、1つのループに結合する。この変換により、ループのオーバーヘッドが減少し、命令オーバーラップを可能にする。キャッシュ競合によるミスを潜在的に増加させ、レジスタ溢れを引き起こす可能性があるが、同一配列を使用するループの結合は、データ局所性を改善できる可能性がある。
- ループタイリング(loop tiling)（またはループブロック化(loop blocking)）は、多重ループの反復空間をブロックに分割する。通常キャッシュメモリの使用効率をよくすることを目的として行われる。
- ループ交換(loop interchange)は、最内並列ループを最外に移すことであるが、データ局所性に逆の影響を及ぼすために注意深く使用する必要がある。（6.11 Improving Data Localityを参照）

次の変換を使用して、タスク数を増やすことができる：

- ループ分散(loop distribution)（またはループ分割（ループスプリット(loop splitting)））：ループ本体を分割し、複数のループを作る。この変換は、ループ本体にある並列計算と逐次計算を分離するためにも使用される。
- ループ一重化(loop coalescing)は、多重ループを1つのループに展開する。

以下は、いくつかの一般的知見である：

1. タスクを大きくすると、通常、よりよいデータ局所性をもたらす。
2. タスクを大きくすると、負荷が不均衡になりやすい。
3. スレッド/タスク数がコア数より多い場合、多くの同期(例えばバリア)や大域的通信が性能を悪化させている。

## 6.6 Improving Load Balancing

スケーラビリティの実現には負荷分散が重要である。タスク実行時間が可変の時（例えば、計算量が入力データ依存の時）、静的スケジューリングでは負荷不均等が起こり得る。負荷均等への最初のステップは、コアの数よりも大幅に多くのタスクに仕事を分割することであ

る。さらに動的タスクスケジューリング戦略（例えばワークスティーリングスケジューリング）を採り、並列計算中、使われていないコアに仕事が割り当てられるようにすべきである。

以下は、いくつかの一般的知見である：

1. 負荷均等の達成は、データ局所性の改善と競合する可能性があり、トレードオフになる場合がある。
2. OpenMPでは並列ループのschedule節において動的スケジューリング戦略を利用できる。
3. 動的タスクスケジューリング戦略は、通常何らかの大域的同期を必要とするため、より多くのオーバーヘッドをもたらし、静的スケジューリング戦略よりスケーラビリティが低くなる。

## 6.7 Removing Synchronization Barriers

バリア同期は、対象スレッドがすべてそのバリアに到達し、与えられた実行ポイントで必要とされるすべての変数が使用可能になるのを保証できるまで、スレッドを待機させる。過剰同期は性能に悪影響を及ぼす可能性があるため、データ依存性の確実な順守が必要な場合や実行頻度が最も低い場所にのみ、バリアを設定すべきである。

以下は、いくつかの一般的知見である：

1. バリア同期を不適切に取り除くと、競合状態が起こる場合がある。
2. 一連のタスク処理において、複数回のスレッド生成及び破棄を一つのバリアに置き換え可能なことがある（すなわち、複数スレッドのジョインおよび生成との置換）。
3. バリアはまた、メモリ内容が最新であることを保証するため、メモリをフラッシュするために使用される。
4. OpenMPなどのプログラミングAPIでは、暗黙のうちにバリアが使われている。

## 6.8 Avoiding Locks/Semaphores

共有するデータまたは部分コードに対し、複数スレッドが同時にアクセスするのを防ぐ（つまり排他制御する）ことがロックの典型的使用法である。ロックは通常、セマフォ（またはミューテックス）を使用して実装される。ロックは逐次化する上にオーバーヘッドが大きいため、可能な限り避けるべきである。ロック使用に関し、主に 2 つの戦略がある：

1. 細粒度ロックでは、低レベルアクセス関数内のアトミックデータ（例えば、データベースの各レコード）をミューテックスを用いて保護する。この方法では、上位の関数はロックを気にする必要がないが、ミューテックスの数が非常に多くなり、デッドロックの危険性が高くなる。
2. 高レベルロックでは、広範囲のデータ（例えば、データベース全体）を単一のミューテックスによって保護する。デッドロックの危険性は低くなるが、領域全体のデータアクセスが逐次化されるため、性能への悪影響が大きくなることがある。

大域的データ構造を複数バケット(bucket)に分割し、バケットごとに個別のロックを使用することで、トレードオフに対応した実装ができ、効率的な分割によって、ロック競合を大幅に減らすことができる。別の方法として、各スレッドが個別に値を計算し、全体の結果に対してのみ同期させる方法がある。

以下に、ロックを回避可能ないくつかの状況を示す：

1. リダクションのような大域的処理のためのロック使用
2. プライベート化できる共有データを排他制御するためのロック使用
3. イベント待ちのために共有変数に対してスピンロックすると、メモリ帯域幅が浪費される可能性がある。

以下は、いくつかの一般的知見である：

1. クリティカルセクションに入る前にノンブロッキングにトライロックする（そしてロックが取れなければ別のジョブに切り替える）ことで、スレッドのアイドル時間を避けることができる。
2. ロックが多すぎると、複雑なデッドロックが発生する可能性がある。
3. いくつかのアーキテクチャは、ロックの置き換えに利用できるアトミックなメモリの読み書き機能を提供する。

## 6.9 Avoiding Atomic Sections

アトミックセクションは一度に 1 つのスレッドだけが実行できる連続した文の集合であり、リダクションを実装するような場合に使われる。アトミックセクションには排他制御と同様の問題がある。

## 6.10 Optimizing Reductions

交換可能で連想的な演算（加算や乗算など）に対する大域的なリダクションは、アトミックセクションを使った単純な実装に代わり、効率的に並列実装できる（Figure 34）。

```
/* File: reduction.c */

// ON ALL CORES/THREADS
void worker1(int *dp, int src1[], int src2[], int bg, int ed) {
 int i,j;
 for (i=bg;i < ed; i++){
  int t;
  t += src1[i] * src2[i];
// START ATOMIC SECTION
  pthread_mutex_lock(dp_mutex);
  *dp += t;
  pthread_mutex_unlock(dp_mutex);
// END ATOMIC SECTION
 }
}

//ALTERNATIVE IMPLEMENTATION

// ON ALL CORES/THREADS
void worker2 (int t[],int src1[], int src2[], int bg, int ed){
 int i,j;
 for (i=bg;i < ed; i++){
  t[MYCOREorTHREADNUMBER] += src1[i] * src2[i];
 }
}

// ON THE MAIN CORE/THREAD
. . .
  for (i=0; i < NBCORE; i++){
   dp = dp + t[i];
  }
. . .
```

**Figure 34. Parallel reductions.**

以下は、いくつかの一般的知見である：

1. ほとんどの並列プログラミングAPIには、リダクションやプレフィクスなどの効率的な演算が含まれる。
2. 並列リダクションは丸め計算を変更する。しかも、プロセッサコア数に依存して結果が変わる可能性がある。

## 6.11 Improving Data Locality

この節では、キャッシュに関する問題について、コヒーレント(coherent、一貫性保証)および非コヒーレントの両方の場合を扱い、説明する。ここで紹介する最適化手法はアプリケーションの範囲に制約があったり、移植性とトレードオフの関係があったりするので、インターフェース設計や運用戦略をよく考えなければならない。4.19 Affinity Scheduling でもデータ局所性やキャッシュ利用最適化に関係する重要な話題が取り上げられているので、そちらも参照してほしい。

### 6.11.1 Cache Coherent Multiprocessor Systems

高いレベルで考えると、マルチコアプロセッサは共有メモリモデルや分散メモリモデルを用いて実装されている。共有メモリシステムのキャッシュメモリは、主記憶の内容のコピーを持つことになる。それゆえ、まるでプロセッサが主記憶と直接接続されているかのように各キャッシュの一貫性（またはコヒーレント）を保つために、キャッシュコヒーレントプロトコルを使用しなければならない。

### 6.11.2 Improving Data Distribution and Alignment

データ分割は、データ配置を改善することによって計算資源利用率や性能を向上さ せるためにも使うことができる。通信コストを減らすためには、あるプロセッサコアが使うデータをそのコアに可能な限り近いところに置くのがよい。言い換えると、処理されるデータはプロセッサコアの「ローカル」に置かれるべきである。これにはデータの再分割、再構築、プリフェッチを実装するためのプログラマの努力が必要になるかもしれない。

マルチコアプロセッサにおいて、共有および分散メモリモデル両方についてデータの分割と配置を考える必要がある。どちらの場合においても、余分な通信を減らすために、主記憶のデータ分割の戦略についてプログラマはよく考えなければならない。

例えば、非アライン(unaligned)の 16 バイトデータ構造について考える。16 バイト幅のコヒーレンス単位（キャッシュライン）を持つ共有メモリプロセッサでは、このデータ構造に対して２回の転送が要求される。この非アラインデータは、DMA の最小転送粒度が 16 バイトのヘテロジニアスマルチコアシステムでも余分な転送を引き起こす。なぜなら、ほとんどのヘテロジニアスマルチコアシステムの DMA 転送は、最小サイズにアラインするためである。

一例として、画像フレームが矩形ブロック単位で処理されるコーデックアプリケーションを考える。この場合、矩形ブロックに対し、アラインされたブロックよりも多くの転送が必要になる。時には、矩形ブロックに対して最小のブロック転送で済むように、プログラマは画像フレーム全体を再構成する。この方法では再構成のオーバヘッドがかかるため、性能とのトレードオフが伴う。小さなスクラッチパッドメモリを持つヘテロジニアスマルチコアシステムに対しては、通信コストを減らすためのソフトウェア手法が提案されている (6.11.5,6)。

プログラマはデータ分割を設計する際、プリフェッチの効果について意識する必要がある。一般に、単一ワードよりも大きな単位でプロセッサにデータが転送されるため、同時に余分なワードが転送される。もしも、他プロセッサによる変更や、要求プロセッサによる置き換えよりも前に、この余分に転送したワードを要求プロセッサが使うならば、追加のデータ転送が不要になる。これを「プリフェッチ効果」という（これはシングルコアにも当てはまるが、マルチコアではその影響がより顕著になる）。

### 6.11.3 Avoiding False Sharing

上述の状況において、もし余分にプリフェッチされたワードが必要でなく、同じキャッシュコヒーレント共有メモリシステム上の他のプロセッサがそのワードを書き換える場合、この余分な転送はシステムの性能や消費エネルギーに悪影響を及ぼす。この現象を「フォルスシェア」とよぶ。

キャッシュコヒーレント共有メモリシステムでは、コヒーレンス処理の発生を減らし、データ局所性を利用する（１回フェッチして何回も再利用する）ために、コアは特定サイズのコヒーレンス単位でメモリを共有する（通常は 32～256 バイト、プロセッサやシステムのアーキテクチャに依存する）。

MSI や MESI[l]のような一般的なキャッシュコヒーレントプロトコルを採用したキャッシュコヒーレントシステムでは、一度に１つのコアだけが特定のキャッシュラインに書き込むことができる。他のコアによってキャッシュされたキャッシュラインのコピーは読み取り専用モード（MESI プロトコルでは’Shared’状態）になっており、書き込み前に無効化しなければならないが、アクセスが読み込みである限りはコア間でキャッシュラインを共有できる。

フォルスシェアとは、実際には共有されていないデータに対し、２つ以上のコア間においてコヒーレンス処理が行われることである。例えば、コア０が書き込もうとしている共有メモリの位置 *x* と、コア１が読み込もうとしている共有メモリの位置 *y* が、同じキャッシュライン（コヒーレンス単位）上に存在したとする。このとき、もし書き込み操作と読み込み操作が交互に行われた場合、システムではフォルスシェアが発生する。

フォルスシェアとそれによる性能低下の影響は以下の３つのステップで説明できる（Figure 35）。1) ワード *x* および *y* を含むキャッシュライン *A* はコア１によって読み込まれ、コア０のキャッシュに *A* のコピーが作られて読み取り専用になる。2)コア０がキャッシュライン *A* に書込み操作を行うとコア１のキャッシュライン *A* は無効化される。3)コア１がワード *y* を読み取ろうとしたとき、コア１で L1 キャッシュミスが発生し、キャッシュライン *A* の最新のコピーがコア１に転送され、コア０のキャッシュライン *A* は読み込み専用に格下げされる。もしワード *x* と *y* が異なるキャッシュライン上にあれば、フォルスシェアによって生じるキャッシュミスを回避できたであろう。

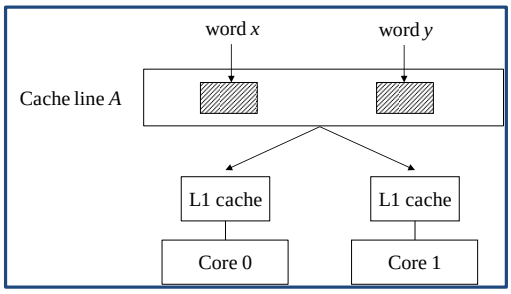



**Figure 35. False sharing situation.**

フォルスシェアは１つ以上のコアが同じキャッシュラインに書き込みを行おうとした場合に発生するため、頻繁に書き込まれるデータが他の関連の薄いデータと同一キャッシュライン上にある時は注意する必要がある。特に、単一の小規模データ構造に複数の同期変数を集約することは避けたほうがよい。分散させることによって可読性を損なう可能性があるが、その代わりに性能向上するだろう。

フォルスシェアを解決する単純な方法は、頻繁に書き込まれる複数のデータが異なるキャッシュラインに配置されるよう、キャッシュラインにデータを足すことである。例えば、 *‘struct test { int x; int y; }’* という構造体を仮定し、コア０によって x が頻繁に書き込まれるとする（Figure 36）。このとき変数 (padding) 追加によって、 メモリ増 を代償にフォルスシェアを避けることができる。

```
struct test

int x;
unsigned char
padding[CACHE_LINE_SIZE -
sizeof(int)];
int y;
```

**Figure 36. Cache-line padding example.**

Figure 36 は概念的な例を示しており、実際のシステム依存アライメント命令を示したわけではない。メモリアライメントのための指示行や *memalign()*のようなライブラリ関数をコンパイラが対応している場合があるが、その記述法や振る舞いはシステム依存である。性能低下やメモリ空間の無駄を避けるためには、特定のコア（スレッド）によってアクセスされるデータを同じあるいは隣接するキャッシュラインに集めるのが望ましい。

プログラマは自動プリフェッチの影響についても考慮する必要がある。これは、最新のプロセッサアーキテクチャで提供されている機能であり、プログラムのアクセスパタンにしたがって隣接キャッシュラインがプリフェッチされるものである。しかし、他の基準を満たしながらフォルスシェアの回避を行うことは必ずしも容易な作業ではない。

### 6.11.4 Cache Blocking (or Data Tiling) Technique

コヒーレントキャッシュにおいては、時間的局所性と空間的局所性を最大化するた めにメモリの再編成を行うことができる。この手法は「キャッシュブロック化(cache blocking)」と呼ばれ、キャッシュに存在するデータの再利用を増やすことによりキャッシュミス率を低下させ、プログラム性能を向上させる。データサイズがキャッシュサイズを超えると、古いデータを追い出して新しいデータを読み込むので、キャッシュミス率が増加する。キャッシュブロック化を使うと、キャッシュに合うサイズのデータを用いて計算できるよう、多重ループが再編成される。

キャッシュブロック化の一例（Figure 37）において、元のループ(Original loop)では合計 4Mbyte の *input_data* 配列の総和計算が 100 回行われる。キャッシュブロック化を適用したループ (Loop with Cache Blocking)では、4Mbyte の *input_data* 配列を 32KB のキャッシュブロック サイズに収まるように分割する。*input_data* のサイズが大きいため、キャッシュヒット率は大幅に向上する。

Original Loop

```
#define TOTAL_SIZE 4194304 /* 4 MB */
#define LOOP_NUM 100
int i, j;
int sum = 0;
int input_data[TOTAL_SIZE] = { 0, 1, };
/* Before cache blocking */
for(j = 0; j < LOOP_NUM; j++)
{
   for(i = 0; i < TOTAL_SIZE; i++)
   {
       sum += input_data[i];
   }
}
```

Loop with Cache Blocking

```
#define TOTAL_SIZE 4194304 /* 4 MB */
#define LOOP_NUM 100
#define CACHE_BLOCK_SIZE 32768 /* CACHE_BLOCK_SIZE divides evenly into
TOTAL_SIZE */
#define BLOCK_NUM (TOTAL_DATA_SIZE/CACHE_BLOCK_SIZE) /* 128 */
int i, j, k;
int sum = 0;
int input_data[TOTAL_SIZE] = { 0, 1, };
/* Cache blocking */
for(k = 0; k < BLOCK_NUM; k++)
{
   for(j = 0; j < LOOP_NUM; j++)
   {
     for(i = k * CACHE_BLOCK_SIZE; i < ( k + 1 ) * CACHE_BLOCK_SIZE;
i++)
     {
     sum += input_data[i];
     }
   }
}
```

**Figure 37. Cache blocking example.**

### 6.11.5 Software Cache Emulation on Scratch-Pad Memories (SPM)

ほとんどのヘテロジニアスマルチコアシステムでは、ハードウェアアクセラレータ（例えばDSP）のキャッシュはコヒーレントではないが、その代わりに、アクセラレータコアにローカルなスクラッチパッドメモリを搭載することが多い。この方法によってエネルギー消費とキャッシュコヒーレンス回路による複雑さを抑えることができる。スクラッチパッドメモリとシステムの主記憶とのデータ転送は、通常 DMA などの手段によって実現される。コアが DMA 転送の完了を待つと性能低下するので、ダブルバッファリングのようなレイテンシ隠蔽手法を使う必要がある（6.1 Performance を参照）。コンパイラを使用してアクセスパタン解析を行い、適切にデータを配置することもできる。これらの方法は連続アクセスやストライドアクセスのような規則的データアクセスパタンに適している。単純な計算では予測できないような間接参照や不規則アクセスパタンでは、これらの方法ではレイテンシを削減できないことがある。

ソフトウェアによりキャッシュをエミュレートすること（以下、ソフトウェアキャッシュ[li]と呼ぶ）は、不規則なアクセスパタンも含め、通信レイテンシを隠蔽する方法である。不規則なアクセスパタンに対して空間的局所性を最大化するため、ソフトウェアキャッシュは、すでにフェッチしたメモリのコピーを再利用のために保持する。つまり、ソフトウェアを使ってハードウェアキャッシュと論理的に同じ動作をするダイレクトマップキャッシュまたは n-way セットアソシアティブキャッシュをエミュレートする。しかし、キャッシュヒットした場合にペナルティがないハードウェアキャッシュと異なり、ソフトウェアキャッシュはキャッシュヒットした場合でも要求されたキャッシュラインがローカルストレージ（スクラッチパッドメモリ）に保持されているかを検査しなければならず、ハードウェアキャッシュに匹敵するパフォーマンスを達成することは難しい。

例えば、ソフトウェアキャッシュを次のように呼ぶとする：

*X = SOFTWARE_CACHE_READ(SYSTEM_MEMORY_ADDRESS)*

*SYSTEM_MEMORY_ADDRESS* に変更が無い場合でも、このコードが実行されるたびにキャッシュヒット検査プログラムが呼び出される。効率的なキャッシュヒット検査機構やアクセスの局所化技術を開発し、キャッシュヒット後に続くアクセスをローカルメモリアクセス（スクラッチパッドメモリアクセス）とすることは性能にとって重要なことである。アクセスパタンをもとにより良い性能を得るために、キャッシュアルゴリズムはアクセスパタンを解析し、将来使われる可能性の高いキャッシュラインをプリフェッチする必要がある。

例として矩形ブロック単位の 2 次元画像処理を考える。2 次元空間的局所性を利用するようなソフトウェアキャッシュを設計可能であり、その結果、矩形ブロックあたり 1 回のキャッシュヒット検査にできる。

ソフトウェア実装であるため、キャッシュ構造（連想率、キャッシュサイズ、インデックス方式）および置換アルゴリズムはコンパイル時または実行時に柔軟に決定できる。これらのパラメタは性能に密接に関係するため、1 つのアプリケーションに対して異なるパラメタを持つ複数のソフトウェアキャッシュを使用することも可能である。

### 6.11.6 Scratch-Pad Memory (SPM) Mapping Techniques at Compile Time

SPM を持つアーキテクチャへのコンパイル時マッピング技術はとても強力であるが、コンパイル時解析が可能なコードを必要とする。アプリケーションのデータ再利用を特定、検出するために配列データフロー解析（幾何学的/多面体解析）が使われるということが最も重要なことである。タスクレベル並列化と組み合わせることで、SPM を持つマルチコアアーキテクチャ向けに、静的解析可能なアプリケーションに対するマッピング解を提供できる。その解では、各コアがアクセスする配列位置を正確に特定し、SPM に配置することにより、フォルスシェアを防ぐことができる。

## 6.12 Enhancing Thread Interactions

不十分な共有資源 (チップ外メモリ帯域など[liii]) やデータ同期オーバーヘッドにより、 マルチコアプロセッサで並列化されたアプリケーションを実行すると、シングルコアプロセッサで実行した場合に比べ性能が落ちることがある。将来のマルチコアプロセッサがより多くのメモリ帯域を持つようになったとしても、コアへのメモリ帯域の配分が適切でない時、システムは性能劣化するだろう。その場合、QoS (Quality of Service)として一般に知られる、 帯域を消費するスレッドの動作を制御する手法の適用を考えることができ、その結果、高い優先度を持ったスレッドがメモリアクセス権を得られることになる。しかし、多くの QoS プロトコルはハードウェア支援を必要とし、もし支援がなければ、プログラマがソフトウェアによってメモリアクセスの振る舞いを制御する必要がある。たとえば、以下の文献では、生産者-消費者(producer-consumer)型のメモリアクセスパターンに対するソフトウェアベースの メモリ制御手法が提案されている。 “M. Alvarez, A. Ramirez, A. Azevedo, C.H. Meenderinck, B.H.H. Juurlink, M. Valero. Scalability of Macroblock-level parallelism for H.264 decoding. The IEEE Fifteenth International Conference on Parallel and Distributed Systems (ICPADS), December 8-11, 2009, Shenzhen, China. http://alvarez.site.ac.upc.edu/papers/parallel_h264_icpads_2009.pdf.”
この手法においては、生産者と消費者間で共有されたキューを監視し、メモリアクセスの意味で高い優先度を持つ生産者タスクにより多くの帯域を割り当てることを目的として、消費者タスクが動作するコア数を制限する。

もし共有キューの長さが短すぎる場合、生産者がメモリ帯域不足になっていると考え、消費者タスクのコア数を減らす。逆に共有キューが長すぎる場合、消費者側の処理を加速させるため

に、消費者タスクのコア数を増加させる。Figure 38 は、並列 H.264 デコードを 4 コアプロセッサ上で実行した場合に共有キューを用いた手法を適用した場合の効果を示している。(実線はこの手法による速度向上を示している。)

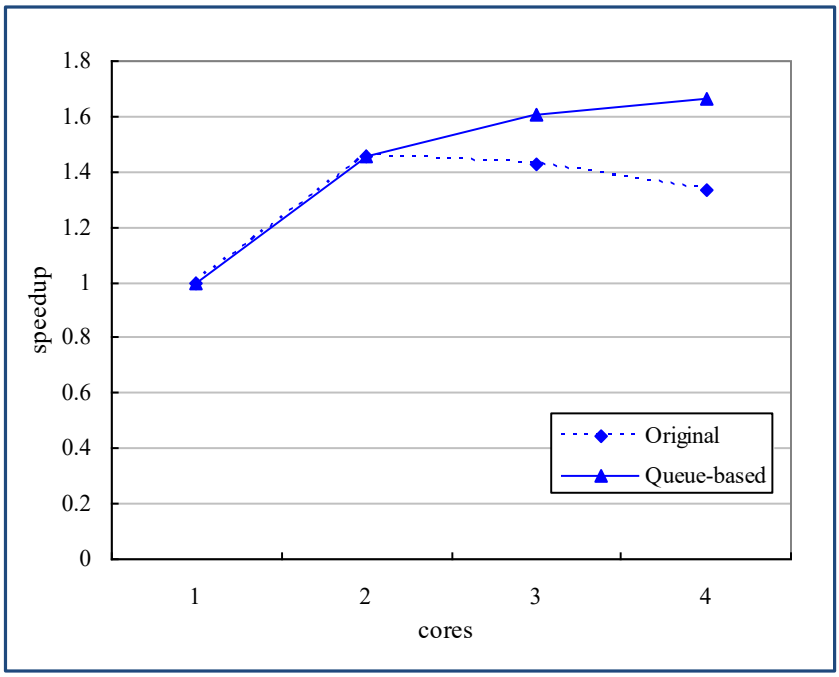



**Figure 38. Performance comparison utilizing queue-based memory access.**

多くのアプリケーションはマルチスレッド化によって恩恵を受けるが、アプリケーション性能が共有データやバス帯域の競合によって制約を受けている場合、スレッドを追加しても性能は向上しない。これらのスレッドはチップ電力を無駄にするばかりである。省電力・高性能実行のためにはスレッド数（と配分）を正しく選択することが重要である。

## 6.13 Reducing Communication Overhead

これまでに通信(例えば、メッセージパッシング)がどれほど並列プログラムの速度を著しく低下させるのか、そして、可能ならば追加の計算コストを費やしてでも避けるべきなのかについて議論してきた。メッセージパッシング時の通信時間 $T_{com}$ は通常次のように表される：

$$T_{com}(n) = \alpha + \beta * n$$

ここで $n$ はメッセージサイズであり、$\alpha$はレイテンシによる開始時間、$\beta$は利用可能な帯域によって決まる単位データ当たりの送信時間である。開始時間は無視できないほど大きいため、実効的な通信帯域を向上させるには、可能ならば、小さなメッセージをできる限りたくさん集め、より大きなメッセージにすることが多くの場合有効である。

以下は、通信に関するいくつかの一般的知見である：

1. 非連続データ送信は通常、連続データ送信に比べ効率が落ちる。
2. 大きすぎるメッセージは絶対に使用しない。APIによって(例えば、いくつかのMPI 実装)はとても大きいメッセージを扱う際はプロトコルを変更することもある。
3. 通信レイテンシとルーティング戦略の要因により、コアへのプロセス/スレッド配置が性能に影響を与えることがある。

## 6.14 Overlapping Communication and Computation

非同期通信やダブルバッファ手法を使って通信と計算をオーバーラップすることで性能向上させることができる(最大 2 倍)。同期(ブロッキング)通信では、通信終了まで送信側に待ちが発生するが、非同期(ノンブロッキング)通信では送信側と受信側が「待ち合わせ（ランデブー(rendezvous)」する必要がない。データ送信は送信者が呼ぶとすぐ開始され、通信が行われている間も引き続き作業を行うことができる。この通信モードでは通信と計算を重ね合わせることができるが、送信データが使用されるまで、送信側か受信側でデータを格納する必要があるため、多くのメモリを必要とする。

以下は、メッセージパッシングに関するいくつかの一般的知見である：

1. 共有メモリを使用したメッセージパッシングは、直接のメモリ間通信より必ずしも優れているわけではない。これは送信と受信の操作の実装に余分なデータコピーが必要になるからである。最先端の実装では、大きなメッセージに対してゼロコピープロトコルを使用することでメモリコピーを避けている。

2. 対応する送信より前に受信準備ができた場合、メッセージパッシングAPIによって(MCAPIなど) はデータを受信バッファに直接格納し、バッファリングするコストを削減することができる。

ダブルバッファは、通信レイテンシを隠蔽するために二つのメモリバッファを使用する。次のデータを片方のバッファに非同期通信によって格納する間、もう一方のバッファに対して計算を実行する。バッファは各ステップで交換される。

## 6.15 Collective Operations

P2P (Point-to-point) 通信は送信者と受信者の二つのタスクを必要とする。多くのタスクが関与する場合、P2P 通信の実装はかなり非効率になる可能性がある。多くの通信 API はタスク集合を扱う集団演算(collective operation)を提供している(Figure 39)。走査(scan)、リダクション（reduction）、ブロードキャスト（broadcast）、分散（scatter）、収集（gather）のような一般的な集団演算が最適化されて提供される。多くの集団演算は少なくとも *log2(P)* (*P* はコア数)のステップを持ち、結果として、それらはコストが高く避けるべき操作となる。

```
/* File: collectiveOperations.c */
//NAÏVE BROADCASTING OF DATA WITH POINT TO POINT MESSAGE
…
if (me == sender){
// start sending data to every other tasks
 for (i=0;i<nbtask;i++){
 if (i != me){
   send(data);
 }
 }
} else {
// receive data from sender
 for (i=0;i<nbtask;i++){
 if (i != sender){
   receive(data);
 }
 }
}

#include "mpi.h"
//BROADCASTING DATA WITH MPI
…
MPI_Bcast (&data, 1, MPI_INT, sender, MPI_COMM_WORLD);
```

**Figure 39. Example code broadcasting data with MPI.**

## 6.16 Using Parallel Libraries

この節では、開発期間を減らし、コードの移植性や再利用性を高め、コード性能を改善する助けになる、並列ライブラリを列挙する。

- ネイティブスレッドライブラリ (訳者注：OSと同じ層で実行されるライブラリをネイティブスレッドライブラリと呼ぶ。Javaや.NETなどのOS上の仮想実行環境で実行されるライブラリをマネージドランタイムシステムと呼び、後述する)
  - Microsoft Win32 and COM Threads
  - POSIX Pthreads
  - Various UNIX, Linux flavors, including Apple MacOS- Wind River VxWorks API, Enea OSE API
- 明示的制御による並列API
  - MPI
  - MCAPI, TIPC, LINX
  - Apple Grand Central fit
- マネージドランタイムシステム
  - Microsoft .NET Task Parallel Library
  - Java Concurrency – java.lang.Thread
  - Python thread library
- スレッド化またはスレッドセーフライブラリ
  - Intel Math Kernel Library
  - Intel Integrated Performance Primitives
- スレッド抽象化
  - Intel Threading Building Blocks
  - OpenMP
  - Microsoft Parallel Patterns Library & Concurrency Runtime
  - Boost thread library
- 理論的（訳者注：原文は Esoteric）
  - Haskell, Erlang, Linda, F#, Cg, HLSL, RapidMind
  - OZ/Mozart
  - CodePlay's C++ compilers
  - Scala
- 並列線形代数ライブラリ
  - ATLAS, BLACS, PBLAS, LAPACK, ScaLAPACK

# CHAPTER 7: FUNDAMENTAL DEFINITIONS

## 7.1 Fundamental Definitions Introduction

マルチコアプログラミングに関わる多様なコミュニティにおいては、一般にそれぞれ独自のボキャブラリや定義があるため、ガイドラインやプラクティスを通して会話することに課題がある。ほとんどのコミュニティは同じボキャブラリを使っているが、残念ながら定義における重なりは小さい。不要な混乱を避けるため、この章では共通的に使われる多くの用語とその定義を示す。本章において用語はアルファベット順に並べられている。また、あまり知られていない可能性があるが、複数定義になりそうにない用語についてもあげてある。

## 7.2 Fundamental Multicore Definitions (Hardware)

本節の定義は、システム構築部品としてのハードウェアに関わるものであり、ソフトウェアやシステムのランタイム構成などは含まない。（訳者注：「＊」の意味については「マルチ＊」を参照）

**アクセラレータ（Accelerator）**：何らかの特定機能を実装した集積回路ブロックのこと。一般にアクセラレータは、同機能を実現するプロセッサコア上のソフトウェア実装よりも高速実行できるよう設計されている。

**非対称ホモジニアスマルチ＊（Asymmetric Homogeneous Multi-*）**：同一 ISA を持つ異なるプロセッサコアとして実装された、複数マイクロアーキテクチャを有するホモジニアスマルチ＊のこと。

**粗粒度マルチスレッド（Coarse-grain Multi-threading (CMT)）**：リソースを共有する複数コンテキストを有するが、一度に一つのコンテキストのみ実行可能なプロセッサコアのこと。複数コンテキスト実行は、サイクル、メモリアクセス、その他の機構による時間分割により実現される。CMT プロセッサの例として Sun UltraSPARC T2 (“Niagara 2”)があげられる。CMT の C は Cooperative や Chip と表されることもあるが、意味は同じである。

**コンテキスト（Context）**：プログラム実行を実現するためにプロセッサコアに含まれるプログラムカウンタおよびレジスタ集合のこと。コンテキストは、ハードウェア関連文書においてはスレッドやストランドとして参照されることも多い。また「論理」プロセッサとして考えられることもある。特権階層（ユーザーレベル、カーネルレベル、ハイパーバイザレベル）内のいくつかのレベルのみ対応するよう定義されることもある。

**ヘテロジニアスマルチ＊（Heterogeneous Multi-*）**：複数 ISA を持つマルチ＊のこと。

**ホモジニアスマルチ＊（Homogeneous Multi-*）**：すべてのプロセッサコアが同一 ISA を持つマルチ＊のこと。ホモジニアスハードウェアということのみを意味しており、ソフトウェアが AMP、SMP、その組み合わせといったような意味は含んでいない。

**メニーコアプロセッサ（Many-Core Processor）**：多数のコアを持つマルチコアプロセッサ（典型的には１６コアより多いもの）のこと。

**マルチ＊（Multi-*）**：マルチコアプロセッサ、マルチコアシステム、マルチプロセッサシステムのこと。マルチ＊システムは、並列システムそのものや並列計算目的で使用される並列システムとして参照されることが多い。

**マルチコアプロセッサ（Multicore Processor）**：２個以上のプロセッサコアを有する半導体チップのこと。

**マルチコアシステム（Multicore System）**：マルチコアプロセッサを有する計算機のこと。

**マルチプロセッサシステム（Multi-Processor System）**：エンドユーザが複数プロセッサコアを利用可能な計算機のこと。

**N ウェイシステム、N コアシステム（N-way System** or **N-core System）**：N 個のコンテキストを有する計算機のことであり、複数のプロセッサもしくは複数コンテキストやコアを持つ単一プロセッサによって構成される。

**N×M ウェイシステム、N×M コアシステム（NxM-way System** or **NxM-core System）**：それぞれのプロセッサが M 個のコンテキストもしくはコアを持つ N 個のプロセッサから構成される計算機のこと。なお、次の論理ステップとして、それぞれ L 個のコンテキストを有するコアをそれぞれ M 個有する N 個のプロセッサから構成される、N×M×L ウェイシステムが考えられるが、使われていない。

**プロセッサ（Processor (CPU)）**：エンドユーザ視点で一つのパッケージとしてみえる、一つ以上のチップとして実装されている一つ以上のプロセッサコアのこと。

**プロセッサコア（Processor Core）**：ユーザがプログラム可能な構成単位として実現された、ある命令セット(ISA)を実現した集積回路ブロックのこと。

**同時マルチスレッディング（Simultaneous Multi-threading (SMT)）**：リソースを共有する複数コンテキストを有し、同時に複数のコンテキストを実行可能なプロセッサコアのこと。単一コア SMT プロセッサとマルチコアプロセッサとは、複数コンテキスト間のリソース共有という意味で異なる。SMT 内の複数コンテキストはリソースを共有するが、マルチコア上の複数コンテキストは共有しない。ただし、どちらもプロセッサコア外でリソースを共有していることがあり得る。SMT プロセッサの例として、ハイパースレッディングを有する

Intel Pentium プロセッサが知られている。マルチコア SMT プロセッサ、すなわち個々が SMT である複数のプロセッサコアを有するプロセッサ、の例としては Intel Core i7 を用いたプロセッサが知られている。

**対称型ホモジニアスマルチ＊（Symmetric Homogeneous Multi-*）**：すべてのプロセッサコアが同一マイクロアーキテクチャ（すなわち同一 ISA でもある）を持つホモジニアスマルチ＊のこと。

**システムオンチップ（System on a Chip (SOC)）**：あるシステム設計に関係する、少なくとも一つのプロセッサと一つの非プロセッサ、例えば計算アクセラレータ、メモリコントローラ、タイマなど、を有する単一半導体チップのこと。（訳者注：原文は single package となっている。複数チップを一つにパッケージングした半導体部品は SiP (System in Package)とよばれ、SoC とは区別されることが多いため、ここではパッケージではなく半導体チップと訳した。）一般に SoC は高度に集積された半導体チップであって、それ自体（例えば外部にメモリを付けるだけで十分など）で（ほぼ）完全なシステムとして動作するものを表す傾向がある。

## 7.3 Fundamental Multicore Definitions (Configuration)

本節の定義は、システムの動的構成上の表現であり、システム内のハードウェア部品やシステム上で動作するソフトウェア部品そのものを意味するものではない。

**ローカルメモリ（Local Memory）**：システムの一部からのみアクセス可能な物理的記憶場所のこと。例として Sony/Toshiba/IBM による Cell BE プロセッサ内の各 SPE に関連付けられたメモリがあげられる。一般にメモリは、プロセッサコア、プロセッサ、SoC、アクセラレータ、ボードなどから局所的アクセスが可能となるため、システムのどの部分がそのメモリにアクセス可能か記述する必要がある。

**共有メモリ（Shared Memory）**：システムの複数部分からアクセス可能な物理的記憶場所のこと。ローカルメモリと同様、システムのどの部分がそのメモリを共有するか記述する必要がある。

## 7.4 Fundamental Multicore Definitions (Software)

本節の定義は、システム上を動作するソフトウェアに関する用語であり、システムのハードウェアやランタイム構成に関わるものではない。

**アプリケーション（Application）**：何かしらの値の計算や、サービスの提供をする一つまたは複数のプログラムのこと。

**非対称型マルチプロセッシング（Asymmetric Multiprocessing (AMP)）**：すべてのプロセッサコアが同様のシステムビューを持たない、もしくは同一主記憶を共有しない、ヘテロジニアスマルチ＊やホモジニアスマルチ＊上で実行される MP もしくは MP かつ MT アプリケーションのこと。

**並行タスク（Concurrent Tasks）**：同時に実行する複数タスクのこと。同時に動作すること以外、それらのタスク間には何も関係がないことを暗に意味している。

**マルチスレッドアプリケーション（Multi-threaded (MT) Application）**：複数スレッドを有するプロセスを少なくとも一つ含むアプリケーションのこと。

**マルチプロセスアプリケーション（Multiprocessing (MP) Application）**：それぞれ固有の仮想アドレス空間を持つ複数のプロセスからなるアプリケーションのこと。これらのプロセスは、OS を使って一つまたは複数のコンテキストを動的に共有し、その上で動作することもあり、あるいは別々のコンテキストに静的に割り当てられることもある。

**並列タスク（Parallel Tasks）**：同一アプリケーション上の並行タスクのこと。

**プロセス（Process）**：同一仮想アドレス空間上の一つまたは複数のコンテキスト上で動作するプログラムのこと。概念的には、一つのプロセスは一つの命令ポインタ、複数のレジスタ値、一つのスタック領域、プロセス存続期間においてアクセス可能な予約済メモリ領域を含む。

**プログラム（Program）**：あるプロセッサ上で動作可能なコンパイル済コードのこと。

対称型マルチプロセッシング（**Symmetric Multiprocessing (SMP)**）：すべてのプロセッサコアが同様のシステムビューと同一の共有主記憶を持つ、ホモジニアスマルチ＊上で実行される MP もしくは MT アプリケーションのこと。

**タスク（Task）**：プロセス内の実行単位のこと。プログラミング手法に依存し、プロセス、スレッド、アクセラレータ使用などがタスクを実行する。

**スレッド（Thread）**：プロセスの一部のことであり、単一コンテキスト上で動作する。

# APPENDIX A: MPP ARCHITECTURE OPTIONS

付録は、オープンソースもしくは製品になっていないソリューションに注目した付加的な情報をまとめたものである。

前提：

- 公開された標準もしくは実装が入手可能である。
- 複数のプラットフォームもしくはＯＳ上の実装が入手可能である。

## A.1 Homogeneous Multicore Processor with Shared Memory

- 多くのAPIが利用可能な広く使われているアーキテクチャである。
- 同一命令セットとデータ保持構造を持つ同一プロセッサコアを用い、プロセス間通信にメモリが用いられる。
- 通常、共有メモリはバスを通してアクセスされ、あるメモリ番地への複数コアによる同時アクセス制御にはロック機構が用いられる。
- 共有メモリは最小オーバーヘッドのコア間通信を容易にし、参照渡しを可能にする。
- 共有メモリに同時アクセスするコア数が増える時、共有メモリはボトルネックとなり得る。このボトルネックにより、非共有メモリアーキテクチャと比較して、共有メモリアーキテクチャのスケーラビリティが悪くなる。
- 共有メモリ型ホモジニアスSMTコアプラットフォーム上では、他のプラットフォームと比較して、スレッドスケジューリングは容易である。
- 共有メモリシステムは配列を利用するマルチスレッドアプリケーション（通常、データを密に共有する）に向いている。そのようなアプリケーションの場合、一般に、共通のキャッシュを持つ隣接CPU上におけるスレッドスケジューリングが容易で、かつ効率が良くなるためである。
- ホモジニアスマルチコアは動的タスクマッピングおよびデータ並列に向いている。
- 共有メモリ型並列プロセッサには、Posix Threads (Pthreads)やOpenMPといった、いくつかの定評ある並列プログラミングモデルが存在する。
- 最近まで、このモデルは組込みデバイスの間では一般的ではなかった。

## A.2 Heterogeneous multicore processor with a mix of shared and non-shared memory

- 共有・非共有メモリ混在型ヘテロジニアスアーキテクチャは、近年まで組込みシステムでは一般的であった。
- 同期のために、コア間メッセージ交換に加え、共有メモリが使われる。
- これらのアーキテクチャは、プログラミングがプラットフォーム依存であるため、学習するのに時間がかかる可能性がある。
- 一般に、一つのプロセッサは主として逐次的で管理的なタスクのために用いられ、その他のコアは同じ種類であり、逐次プロセッサへの通信経路と、自分自身のメモリネットワークを持つ。これらのシステムは歴史的にはマルチメディア計算から発展してきており、それぞれが固有のプログラミング環境を持っている。

## A.3 Homogeneous Multicore Processor with Non-shared Memory

- このアーキテクチャは、一般に、いくつかの箇所において小規模データのみを共有する比較的独立したスレッドを持つようなデータ並列アプリケーションに向いている。
- このモデルはHPC (high performance computing)クラスタにおいて一般的である。

## A.4 Heterogeneous Multicore Processor with Non-shared Memory

- 最近まで、このアーキテクチャは組込みシステムにおいてよくあるものだった。
- これらは一般に用途向けに作られた応用特化型プロセッサの集合体であり、それぞれのプロセッサは個別の命令セットとデータ保持構造を持つ。
- これらのシステムでは一般に静的なタスクマッピングが用いられる。
- コア間通信にはメッセージパッシングが用いられる。

## A.5 Heterogeneous Multicore Processor with Shared Memory

- このアーキテクチャの選択および利用法は、それぞれの産業やターゲットアプリケーションごとに異なっている。
- コアの異種性により、異なる命令セット、データ保持構造を持つことになるため、複雑性を増加させる可能性がある。

# APPENDIX B: PROGRAMMING API OPTIONS

前提：

- 公開された標準もしくは実装が入手可能である。
- 複数のプラットフォームもしくはＯＳ上の実装が入手可能である。

## B.1 Shared Memory, Threads-based Programming

### B.1.1 Pthreads (POSIX Threads)

- スレッドの標準であり、スレッド生成、操作、管理、ミューテックスやシグナルを用いたスレッド間同期に対するAPIを規定している。
- C言語における型と関数の集合として定義されている。*pthread.h* という名のヘッダ/includeファイルとライブラリによって実装されている。
- LinuxやSolarisといったUnix系のPOSIXシステムにおいて実装されている。Pthreads APIの一部はMicrosoft Windowsでも提供されている (pthreads-w32) 。

### B.1.2 GNU Pth (GNU Portable Threads)

- Unixプラットフォームのための POSIX/ANSI-Cに則ったライブラリである。
- イベントドリブンアプリケーションの内部において非プリエンプティブで優先度ベースのマルチスレッドスケジューリングを提供する。各スレッドは、固有のプログラムカウンタ、ランタイムスタック、シグナルマスク、errno (error number)変数を持つ。またいくつかのPthreads APIも提供されている。
- スレッドは、優先度とイベントベースの非プリエンプティブスケジューラによって管理される。また、カーネル空間スレッドに対してM:1対応のマッピングを使っている。ユーザ空間上のM個のスレッドに対しては、スケジューリングはGNU Pthライブラリによって行われ、カーネルは関与しない。このことはSMP、。の利用を不可能にしている。SMPではカーネルによるディスパッチが必要であるためである。
- GNU Pthは広く使われているとは言えない。

### B.1.3 OpenMP (Open Multiprocessing)

- 共有メモリ型の複数のアーキテクチャ、プラットフォーム上において、C/C++/Fortran言語における並列処理プログラムを実現できるAPI。
- コンパイラ指示行、ライブラリ関数、ランタイム環境変数からなる。
- Fork-join並列化を利用する。マスタースレッドが必要に応じて複数のスレッドを産み出す。順序関係をつける、あるいは共有データアクセスから保護するために同期を使う。
- グローバル変数はスレッド間で共有される。並列領域からよばれた関数/サブプログラム内のスタック変数およびステートメントブロック内の自動変数はプライベートである。
- OpenMPを利用するためにはOpenMP準拠のコンパイラ、スレッドセーフなライブラリ関数が必要である。
- OpenMPを利用するために、OpenMPを認識するデバッガを必要とする。そのようなデバッガによってOpenMPの構造や型に関連した情報にアクセスできる。

### B.1.4 Threading Building Blocks (TBB)

- Intel Threading Building Blocksとして知られている。
- 移植可能なC++テンプレートのライブラリである。高レベルのタスク並列であり、性能やスケーラビリティに対するプラットフォームの詳細やスレッド機構が抽象化されている。
- テンプレートを使うことによりコンパイル時のポリモルフィズムが利用できる。
- プロセッサコア間の並列負荷均衡のためタスクスティールが実装されている。
- 処理をタスクとして扱い、ランタイムによってタスクの動的コア割当を行うことによって複数プロセッサを利用するためのデータ構造とアルゴリズムが組み込まれている。

### B.1.5 Protothreads (PT)

- メモリ制約の強いシステムのために設計された超軽量、スタックの無いスレッドである。Ｃ言語で実装されたイベントドリブンシステムのための線形コード実行を提供する。（訳者注：線形コード実行の意味は、下に出てくる逐次制御フローと同義と思われる。）
- ブロックするイベントハンドラを前提としているため、ＯＳ有でも無でも使うことができる。逐次制御フローのみが提供され、複雑な状態機械や完全なマルチスレッド動作をしない。（訳者注：すなわちスレッドスケジューラのようなものがない。スタッ

クはスレッドごとではなく全体で一つしかなく、コンテキストスイッチはスタックの上下だけで実現する。）

- Protothreadsでは、スレッド内独自データを持つことはできず、スケジューリングも手動で行われる。スレッドスケジュールは*thread*関数の呼び出しによって実現され、スレッド状態が適切に管理される。

## B.2 Distributed Memory, Message-Passing Programming

### B.2.1 Multicore Communications API (MCAPI)

- 近接して分散する組込みシステムにおける、近接して分散するコアもしくはプロセッサ間の同期通信に対するメッセージパッシングAPIの標準。ターゲットシステムは様々な異種性を持つことができる。

### B.2.2 Message Passing Interface (MPI)

- 広く利用可能なメッセージパッシングライブラリの標準であり、高並列計算機やワークステーションクラスタにおいて高性能を達成するために設計された。2点間通信や集団通信を実現するためのメッセージパッシングAPIと、その実装における関数のプロトコルとセマンティックの仕様を提供している。
- 基本的な仮想トポロジ、同期、プロセス間通信を言語非依存の形で提供すると共に、言語との結合方法、言語依存の特徴についても与えている。
- 各ノードで同じプログラムを実行するためには陽に示された制御論理が必要である。

### B.2.3 Web 2.0

- Web 2.0基盤は、サーバソフト、コンテンツの要約配信、メッセージのプロトコル、プラグインや拡張機能を含む標準に従うブラウザ、様々なクライアントアプリケーションを含んでいる。
- Web 1.0機能の反復利用を土台としたWeb 2.0により、ユーザーはブラウザを通してソフトウェアアプリケーションを実行できる“network as platform”を利用することができるようになる。

## B.3 Platform-specific Programming

### B.3.1 Open Computing Language (OpenCL)

- ヘテロジニアスプラットフォーム上で実行するプログラムのためのC言語ベースのフレームワークである。カーネル（GPUプログラム等で使われる意味でのカーネル）を記述するための言語とヘテロジニアスプラットフォームを定義、制御するためのAPIを含む。ハードウェアと数値精度要求を定義する。
- タスク並列とデータ並列を利用でき、整合性が緩和された共有メモリモデルが実装できる。重ね合わせることができる異なるアドレス空間を記述できる。
- この標準は今後長い時間をかけて成熟していくことが期待されている。

# APPENDIX C: PARALLEL PROGRAMMING DEVELOPMENT LIFECYCLE

## C. 1 Introduction to Parallel Tool Categories

既存の逐次アプリケーションをマルチコアプラットフォームに移行したいプログラマにとって、数多くの支援ツールが利用可能である。Figure 40 は種々のツール間連携フローを書いた高レベルの図である[liii]。まず、アプリケーションにおいて、逐次、並列双方において性能に影響のあるボトルネックは、並列化前に取り除いておくべきである。スレッド API に関する情報については付録Ｂに記載した。以下では Figure 40 に示しているツール分類について簡単に紹介する。

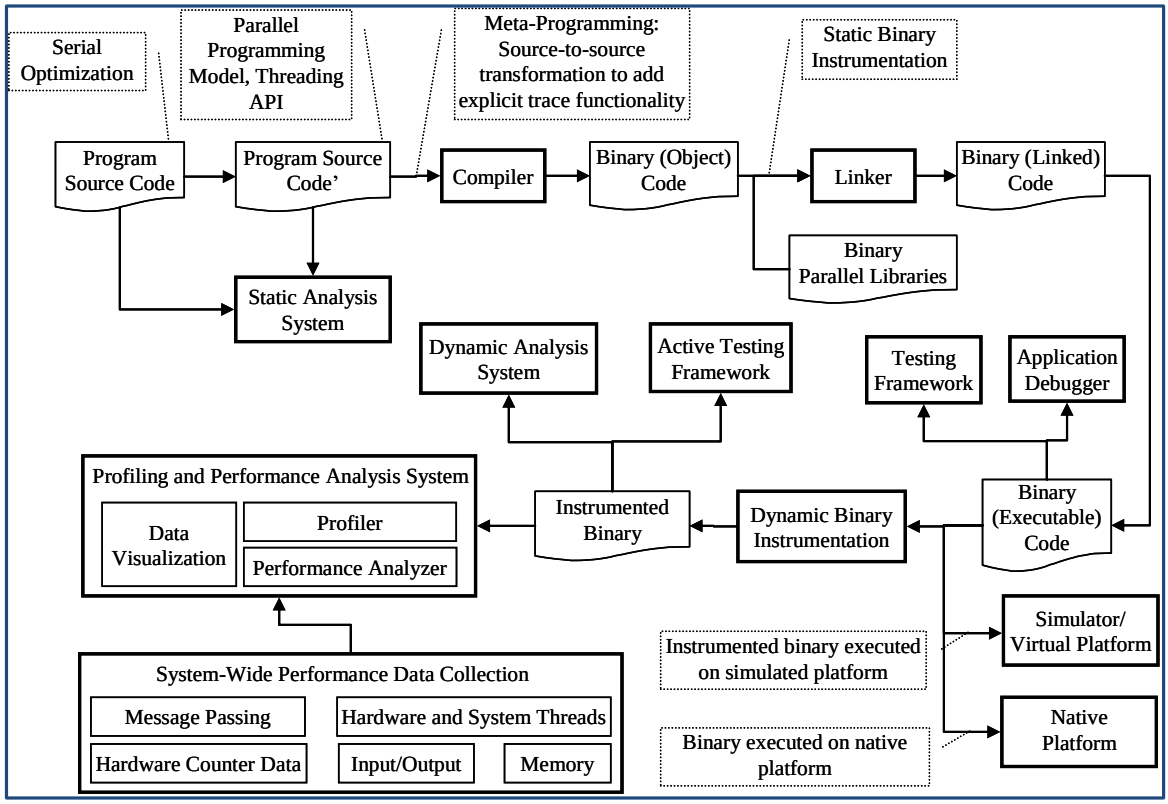



**Figure 40. Workflow between the various tool categories.**

## C.1.1 Compilers

スレッド API に依存するものではあるが、数多くのコンパイラが利用可能である。Figure 40 に示しているように、コンパイラはトレース情報を生成する目的で、ソース計測化、静的バイナリ計測化、動的計測化によってコードを変更することがある。ソース計測化は、プリプロセスやコンパイルの前にソースコードを修正する。静的バイナリ計測化はコンパイル済バイナリを実行前に修正する。

## C.1.2 Static Code Analyzers

静的コード解析は、機能テスト時に見過ごされる可能性がある問題を見つけるため、アプリケーションを実行することなくソースコードを解析する。解析ツールは、ソースコードを使ってソフトウェアアプリケーションのモデルを考え、すべての実行パスを網羅的に探索する。引数の数や型の誤り、無限ループ、実行されないコード、初期化されない変数やポインタ、

配列の範囲外参照、変数や構造体への誤ったアクセスなどが典型的な検出エラーの例である。静的コード解析をビルドプロセスに統合することは、開発プロセスにおける早期問題発見に役立つため推奨される。

### C.1.3 Debuggers

非決定的なスレッドスケジューリングやプリエンプション、制御フローとデータフロー間の依存に起因する複雑性によって、マルチスレッドアプリケーションのデバッグはより難しくなる傾向にある。その上、デッドロックや競合といった、スレッド間干渉によって引き起こされる問題はデバッガの使用によって再現しなくなることもある。

並行システムのためのデバッガとして、従来型デバッグ手法やイベントベースデバッグ手法を含む様々なデバッグ手法を利用できる可能性がある。従来型デバッグ手法（ブレークポイントデバッグ）では、並列プロセスごとに一つの逐次デバッガを使う方法があるが、各デバッガから得られる情報が制限される。イベントベース（モニタ）デバッガはマルチスレッドアプリケーションに対する再現性は良くなるが、オーバーヘッドが大きくなる。スレッドAPIの選択が、デバッガの選択に影響する。

### C.1.4 Dynamic Binary Instrumentation

動的バイナリ計測化（Dynamic binary instrumentation (DBI)）はバイナリアプリケーションの実行時の振る舞いを見出すために使われる。計測コードが挿入され、アプリケーションの命令列の一部として実行される。DBIは実行されたコードパスのみを調べることができる。二つのアプローチ、軽量DBIと重量DBIが知られている。軽量DBIではアーキテクチャ依存の命令列と状態が利用される。一方、重量DBIでは命令列と状態を抽象化する。軽量DBIと比べ、重量DBIの方がアーキテクチャ間での移植が容易である。

### C.1.5 Dynamic Program Analysis

動的プログラム解析は実行時にプログラムの特性を調べるものであり、大規模な負荷テストよりも大幅に早い問題発見の助けとなる。動的解析はハードウェアや仮想プロセッサを使って行われる。これらのツールは一般に自動化が容易で、誤検出(false positive)率が低くなる。動的解析の効率をあげるためには、優れたコードカバレッジを持つテスト入力を使うと良い。

### C.1.6 Active Testing

アクティブテストは静的および動的コード解析を利用して並行性に関わる問題（アトミック違反、データレース、デッドロック）を見つける方法である。見つかった問題の可能性に対

し、スケジューラの入力として与える（当該スケジュールを実現する）ことにより、解析中に見つかる並行性関連問題の誤検出(false positive)を最小化する。

### C.1.7 Profiling and Performance Analysis

プロファイラは、実行プログラムの振る舞いを調べることにより、システムリソースの効率的利用やプログラム分割最適化を容易にするために利用される。また、性能および実行に影響がありそうな問題を発見するためにも用いられる。能動的プロファイラと受動的プロファイラがある。能動的プロファイラは測定対象システムからトレース収集エンジンをコールバックすることにより実行時の振る舞いを記録する。受動的プロファイラは外部要素、例えばプローブや変更した実行環境、を用い、制御フローや実行状態を調べる。一般に、受動的プロファイルでは、測定対象システム自体の変更は必要ではない。

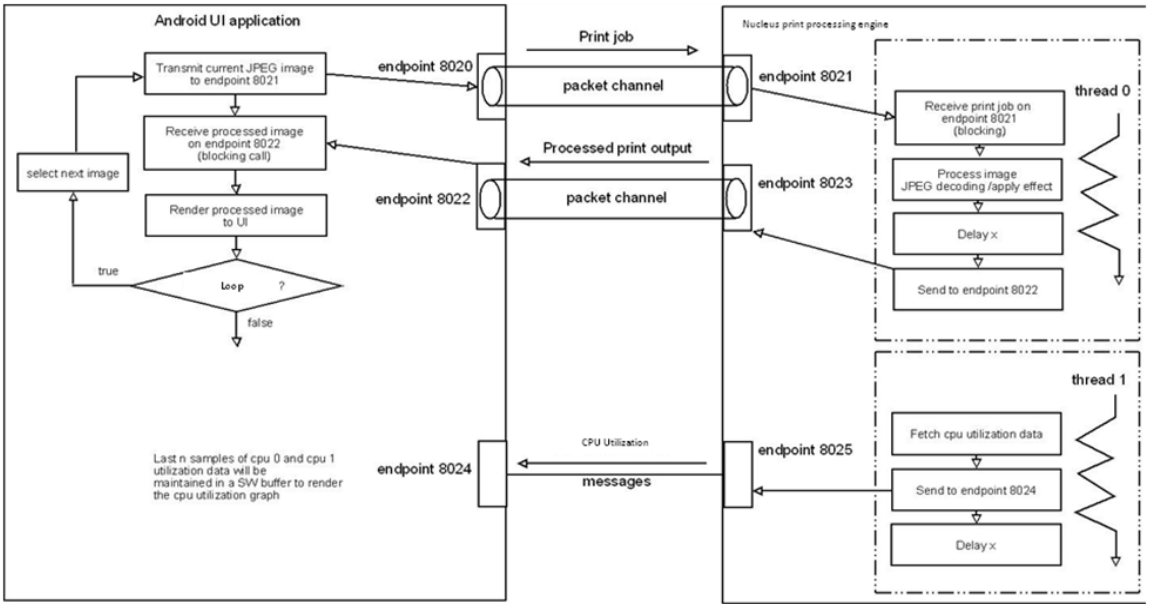
プロファイラからは、実行制御パスに対する振る舞いデータが提供される。注意深く選ばれた入力データや人為的な故障注入を用い、複数のアプリケーションを実行することによりコードカバレッジを向上させることができる。

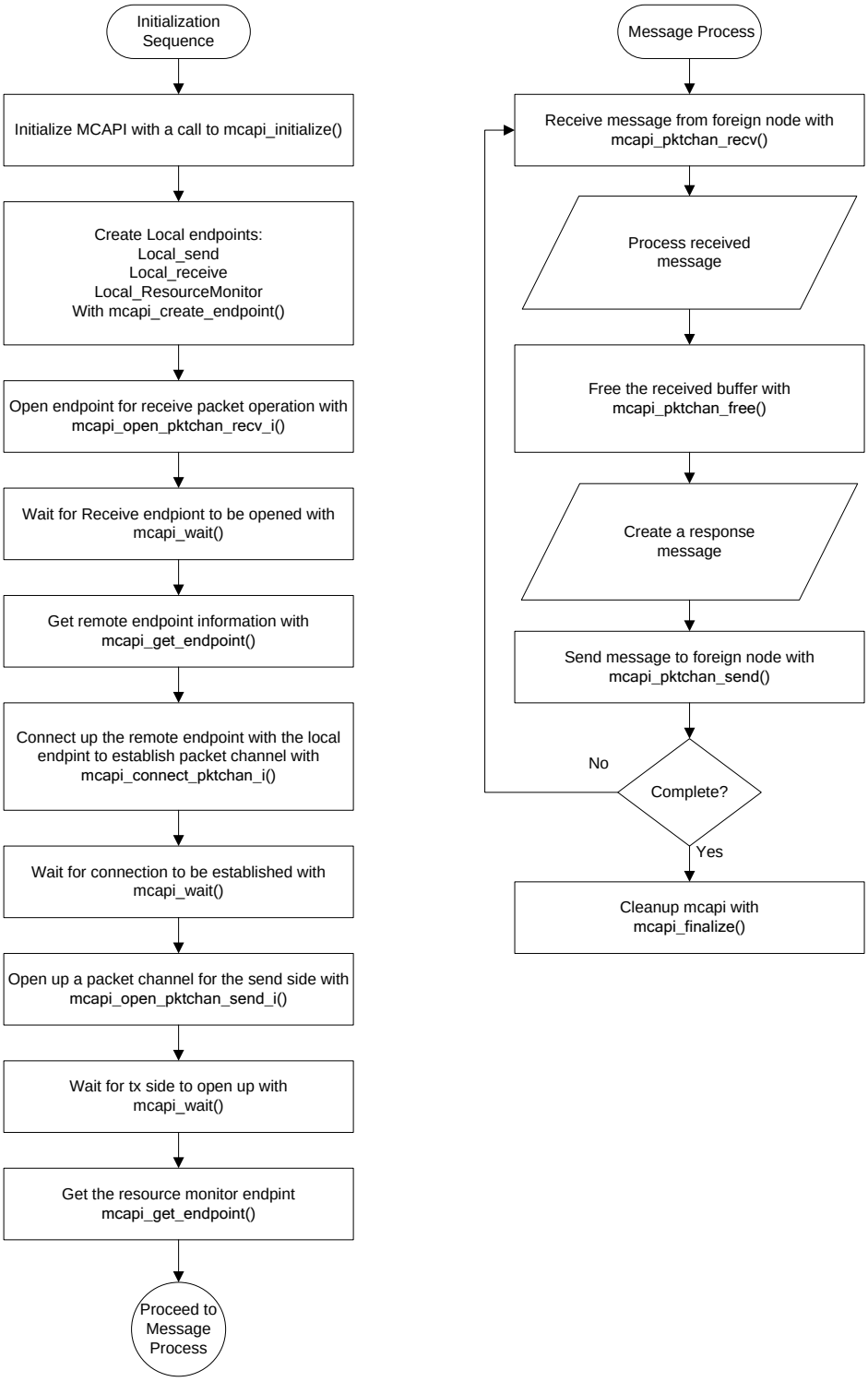
プロファイラを選ぶ際、プログラマは、プロファイラが特定のハードウェア支援を必要とするか、システム全体にわたるプロファイルかユーザアプリケーションにのみ注目したものか、理解しておく必要がある。また、プロファイラは特定のシステムリソース、例えばスレッドプロファイル、コールスタック、メモリプロファイル、キャッシュプロファイル、ヒーププロファイルなどの利用解析のために設計されている場合もある。

### C.1.8 System-wide Performance Data Collection

システムリソースを最大限利用するためには、プログラマはシステム全体にわたるリソース利用データにアクセスし、その情報とアプリケーション性能とを関係付けできる必要がある。性能解析機構やプロファイラが与える低レベルのシステムデータを得るために、標準的 API が役に立つこともある。

# APPENDIX D


Android UI application
Transmit current JPEG image to endpoint 8021
Receive processed image on endpoint 8022 (blocking call)
Render processed image to UI
Loop ?
true
false
select next image
Last n samples of cpu 0 and cpu 1 utilization data will be maintained in a SW buffer to render the cpu utilization graph
endpoint 8020
endpoint 8022
endpoint 8024
Print job
packet channel
Processed print output
packet channel
CPU Utilization
messages
Nucleus print processing engine
endpoint 8021
endpoint 8023
endpoint 8025
thread 0
Receive print job on endpoint 8021 (blocking)
Process image JPEG decoding /apply effect
Delay x
Send to endpoint 8022
thread 1
Fetch cpu utilization data
Send to endpoint 8024
Delay x


Initialization Sequence
Initialize MCAPI with a call to mcapi_initialize()
Create Local endpoints:
Local_send
Local_receive
Local_ResourceMonitor
With mcapi_create_endpoint()
Open endpoint for receive packet operation with mcapi_open_pktchan_recv_i()
Wait for Receive endpiont to be opened with mcapi_wait()
Get remote endpoint information with mcapi_get_endpoint()
Connect up the remote endpoint with the local endpint to establish packet channel with mcapi_connect_pktchan_i()
Wait for connection to be established with mcapi_wait()
Open up a packet channel for the send side with mcapi_open_pktchan_send_i()
Wait for tx side to open up with mcapi_wait()
Get the resource monitor endpint mcapi_get_endpoint()
Proceed to Message Process
Message Process
Receive message from foreign node with mcapi_pktchan_recv()
Process received message
Free the received buffer with mcapi_pktchan_free()
Create a response message
Send message to foreign node with mcapi_pktchan_send()
No
Complete?
Yes
Cleanup mcapi with mcapi_finalize()

```
/***************
* Include Files
****************/
…
/* IPC */
#include          "mcapi/inc/mcapi.h"

/**************
IPC connection
***************/

/* MCAPI definitions */

#define           LOCAL_NODEID            1
#define           REMOTE_NODEID           0

static const struct


[**UNRESOLVED****UNRESOLVED****UNRESOLVED****UNRESOLVED**]ports[2] = {
[**UNRESOLVED**],[**UNRESOLVED**]};

mcapi_pktchan_recv_hndl_t     send_handle;
mcapi_pktchan_recv_hndl_t     recv_handle;
mcapi_endpoint_t              local_rm_endpoint;
mcapi_endpoint_t              remote_rm_endpoint;

/*======================================================================
*
*   FUNCTION
*
*       Initialization
*
*   DESCRIPTION
*
*       This task initializes the IPC.
*
*======================================================================*/
void  Initialization()


      {mcapi_version_tmcapi_version;mcapi_status_tmcapi_status;mcapi_endpoint_tloca
l_send_endpoint;mcapi_endpoint_tlocal_recv_endpoint;mcapi_endpoint_tremote_recv_end
point;mcapi_request_trequest;size_tsize;/* Initialize MCAPI
*/mcapi_initialize(LOCAL_NODEID, &mcapi_version, &mcapi_status);/* Create a local
send endpoint for print job processing. */if(mcapi_status ==
MCAPI_SUCCESS)[**UNRESOLVED**]

      /* Create a local receive endpoint for print job processing. */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Create a local send endpoint for resource monitor. */
      if(mcapi_status == MCAPI_SUCCESS)
```

```
[**UNRESOLVED**]

      /* Open receive side */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Wait for the rx side to open. */
      mcapi_wait(&request, &size, &mcapi_status, 0xFFFFFFFF);

      /* Wait till foreign endpoint is created */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Connect node 0 transmitter to node 1 receiver */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**r]

      /* Wait for the connect call to complete. */
      mcapi_wait(&request, &size, &mcapi_status, 0xFFFFFFFF);

      printf("Connected \r\n");

      /* Open transmit side */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Wait for the tx side to open */
      mcapi_wait(&request, &size, &mcapi_status, 0xFFFFFFFF);

      /* Get the remote resource monitor endpoint */
      remote_rm_endpoint = mcapi_get_endpoint(REMOTE_NODEID,
ports[REMOTE_NODEID].rm_rx, &mcapi_status);

      if(mcapi_status == MCAPI_SUCCESS)


[r**UNRESOLVED**]
}
```

```
/*********************************************************************
*
*   FUNCTION
*
*       Sample
*
*   DESCRIPTION
*
*       This task receives a job and responds to job over MCAPI.
*
*
*********************************************************************/
void  Sample()


      {STATUSstatus;mcapi_status_tmcapi_status;UINT32size;UINT32bytesReceived,type,
cmd;unsigned char *in_buffer;unsigned char
out_buffer[MAX_SIZE];mcapi_uint64_ttmp;/* Receive image from foreign node
*/mcapi_pktchan_recv(recv_handle, (void **)in_buffer,(size_t*)&size,
&mcapi_status);/* Do something with MCAPI buffer */… /* Free MCAPI buffer
*/if(mcapi_status == MCAPI_SUCCESS)[**UNRESOLVED**]

      …
      /* Respond to message with data to send in out_buffer */
      mcapi_pktchan_send(send_handle, out_buffer, MAX_SIZE, &mcapi_status);
      if(mcapi_status != MCAPI_SUCCESS)


[n]

      if ( /* Complete then clean up */ )


            {/* Finalize current nodes MCAPI instantiation
*/mcapi_finalize(&mcapi_status);if(mcapi_status != MCAPI_SUCCESS)[n]
      }
}
```

## ろ



## わ

# References

[i] 'The Open Group Base Specifications Issue 6, IEEE Std 1003.1'によって仕様化された POSIX スレッドは、製品または研究のいずれでも非標準の拡張機能は使っていない。

[ii] 製品または研究のいずれでも非標準の拡張機能は使っていない。http://www.multicore-association.org/workgroup/mcapi.php

[iii] ホモジニアスマルチコアアーキテクチャの中には、共有・非共有混在型のローカルメモリを持つものがあるが、本文書ではこれらについては考えない。

[iv] 本節を通して、マルチコアに移行する主たる目的は（開始から終了までの時間）性能改善と想定する。もちろん、例えば消費電力改善のような他の移行理由もある。本節のガイドラインの多くは異なる改善目標に適用可能であるが、本文中で説明する例では、主たる指標としては性能改善を用いている。

[v] Donald Knuth. "Structured Programming with go to Statements." Computing Surveys, vol 6, no 4, Dec 1974, p 268.

[vi] 並列化を開始すると計算機パラメタの選択も重要になる。異なる環境における変化の効果についてよく理解するために、様々なアーキテクチャ上で複数のコア数を選択するのがよい。特に、長期間使うプログラムは様々な環境で実行できることが期待されている。様々な環境でのブログラム動作を理解することは、特定の環境で過剰に最適化することを防ぐことにもつながる。

[vii] Grotker、Holtmann、Keding、Wloka は、調べたい部分の測定に対し、測定結果内のばらつきの影響をなくすため、十分長い時間、最低でもサンプリング間隔の 2 桁長い実行時間、測定することを推奨している。

[viii] これは逐次性能チューニングにおける一般的な手法であるが、これだけが唯一の方法ではない。時には、効果の大きい単一のホットスポットを抜き出すことが難しい場合もある。そのような場合には、プログラム内の小さな非効率箇所を多数見つけ、「あらゆる場所を磨く」ことが可能かもしれない。

[ix] これはマルチ・メニープロセッサシステム、マルチコアシステムの話であり、従来の HPC 分野では考え方が異なるかもしれないが、HPC は本ガイドの注目領域ではない。

[x] ここでは、プロセッサ稼働率を保つための特殊なワークキューやスレッドプールを使っていないことを想定している。

[xi] http://www.threadingbuildingblocks.org/

[xii] http://en.wikipedia.org/wiki/Sobel_operator

[xiii] R. Johnson, Open Source POSIX Threads for Win32, Redhat, 2006.

[xiv] D.B. Bradford Nichols and J.P. Farell, Pthreads Programming, O'Reilly & Associates, 1996.

[xv] C.J. Northrup, Programming with UNIX Threads, John Wiley & Sons, 1996.

[xvi] D.R. Butenhof, Programming with POSIX Threads, Addison-Wesley, 1997.

[xvii] G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Prentice Hall, 2006.

[xviii] G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Prentice Hall, 2006.

[xix] E.W. Dijkstra, "Cooperating Sequential Processes", (Techniche Hogeschool, Eind- hoven, 1965) Reprinted in: F. Genuys (ed.), Programming Languages, vol. 43, 1968.

[xx] G.L. Peterson, "Myths about the Mutual Exclusion Problem", Information Processing Letters, vol. 12, 1981, p. 115-116.

[xxi] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", Communications of the ACM, Jan. 1965.

[xxii] J.H. Anderson, "A Fine-Grained Solution to The Mutual Exclusion Problem", Acta Informatica, vol. 30, 1993, p. 249-265.

[xxiii] M. Raynal, Algorithms for Mutual Exclusion, MIT Press, 1986.

[xxiv] J.H. Anderson, "Lamport on mutual exclusion: 27 years of planting seeds", PODC: 20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, vol. 12, 2001, p. 3-12.

[xxv] G.L. Peterson, "Myths about the Mutual Exclusion Problem", Information Processing Letters, vol. 12, 1981, p. 115-116.

[xxvi] K. Alagarsamy, "Some Myths About Famous Mutual Exclusion Algorithms", SIGACT News, vol. 34, Sep. 2003, p. 94-103.

[xxvii] T.A. Cargill, "A Robust Distributed Solution to the Dining Philosophers Problem", Software---Practice and Experience, vol. 12, Oct. 1982, p. 965-969.

[xxviii] M. Suess and C. Leopold, "Generic Locking and Dead-Prevention with C++", Parallel Computing: Architectures, Algorithms and Applications, vol. 15, Feb. 2008, p. 211-218.

[xxix] H.M.S.D.K. Chen and P.C. Yew, "The Impact of Synchronization and Granularity on Parallel Systems", Proceedings of the 17th International Symposium on Computer Architecture, June. 1990, p. 239-249.

[xxx] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era", IEEE Computer, Jan. 2008, p. 1-6.

[xxxi] V. Sarkar, Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors, London, UK: Pitman, 1987.

[xxxii] O. Sinnen, Task Scheduling for Parallel Systems, NJ: John Wiley & Sons, 2007.

[xxxiii] M. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, 1979.

[xxxiv] J. Beck and D. Siewiorek, "Automated Processor Specification and Task Allocation for Embedded Multicomputer Systems: The Packing-Based Approaches", Symposium on Parallel and Distributed Processing (SPDP '95), Oct. 1995, p. 44-51.

[xxxv] J.E. Beck and D.P. Siewiorek, "Modeling Multicomputer Task Allocation as a Vector Packing Problem", ISSS, 1996, p. 115-120.

[xxxvi] K. Agrawal, C. Leiserson, Y. He and W. Hsu, "Adaptive Work-Stealing with Parallelism Feedback", Transactions on Computer Systems (TOCS, vol. 26, Sep. 2008.

[xxxvii] M. Michael, M. Vechev and V. Saraswat, "Idempotent Work Stealing", PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, Feb. 2009.

[xxxviii] Shameem Akhter, Jason Roberts, “Multi-Core Programming: Increasing Performance through Software Multithreading,” Intel Press, 2006.

[xxxix] Daniel G. Waddington, Nilabya Roy, Douglas C. Schmidt, “Dynamic Analysis and Profiling of Multi-threaded Systems,” http://www.cs.wustl.edu/~schmidt/PDF/DSIS_Chapter_Waddington.pdf, 2007.

[xl] Yusen Li, Feng Wang, Gang Wang, Xiaoguang Liu, Jing Liu, “MKtrace: An innovative debugging tool for multi-threaded programs on multiprocessor systems,” Proceedings of the 14th Asia Pacific Software Engineering Conference, 2007

[xli] Domeika, Max, “Software Development for Embedded Multicore Systems,” Newnespress.com, 2008.

[xlii] Brutch, Tasneem, “Migration to Multicore: Tools that can help,” USENIX ;login:, Oct. 2009.

[xliii] M. Naik, C. S. Park, K. Sen, and D. Gay, “Effective Static Deadlock Detection”, ICSE 2009, Vancouver, Canada.

[xliv] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, “Using Static Analysis to Find Bugs”, IEEE Software, Sept. 2008.

[xlv] Domeika, Max, “Software Development for Embedded Multicore Systems,” Newnespress.com, 2008.

[xlvi] Valgrind Instrumentation Framework website, http://valgrind.org/.

[xlvii] Pallavi Joshi, Mayur Naik, Chang-Seo Park, Koushik Sen, “CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs,” http://berkeley.intel-research.net/mnaik/pubs/cav09/paper.pdf.

[xlviii] Thorsten Grotker, Ulrich Holtman, Holger Keding, Markus Wloka, “The Developer’s Guide to Debugging,” Springer, 2008.

[xlix] U. Banerjee, B. Bliss, Z. Ma, P. Petersen, “A Theory of Data Race Detection,” International Symposium on Software Testing and Analysis, Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging, Portland, Maine, http://support.intel.com/support/performancetools/sb/CS-026934.htm

[l] MSI (Modified, Shared, Invalid) と MESI (Modified, Exclusive, Shared, Invalid) プロトコルはキャッシュが持つ状態によって名前が付けられており、共有メモリ型マルチプロセッサシステムにおいてよく利用されている。これらのプロトコルでは、書き込みアクセスを逐次化するため、キャッシュラインの"Exclusive" や "Modified"の一意性を保証することにより、キャッシュの*一貫性（コヒーレント）*を保っている。これらのプロトコルの派生形である、MOSI や MOESI についても研究されている。

[li] ソフトウェアキャッシュは、ソフトウェア制御ハードウェアキャッシュと明確に区別する必要がある。ソフトウェア制御ハードウェアキャッシュとは、フラッシュやコヒーレンスの制御をソフトウェアで行い、その他の操作はハードウェアで行うものである。

[lii] ここでは、データ局所性に関する他の実行可能な手法はすべて適用済と仮定している。

[liii] Brutch, Tasneem, “Parallel Programming Development Life Cycle: Understanding Tools and their Workflow when Migrating Sequential Applications to Multicore Platforms,” USENIX ;login:, Oct. 2009.