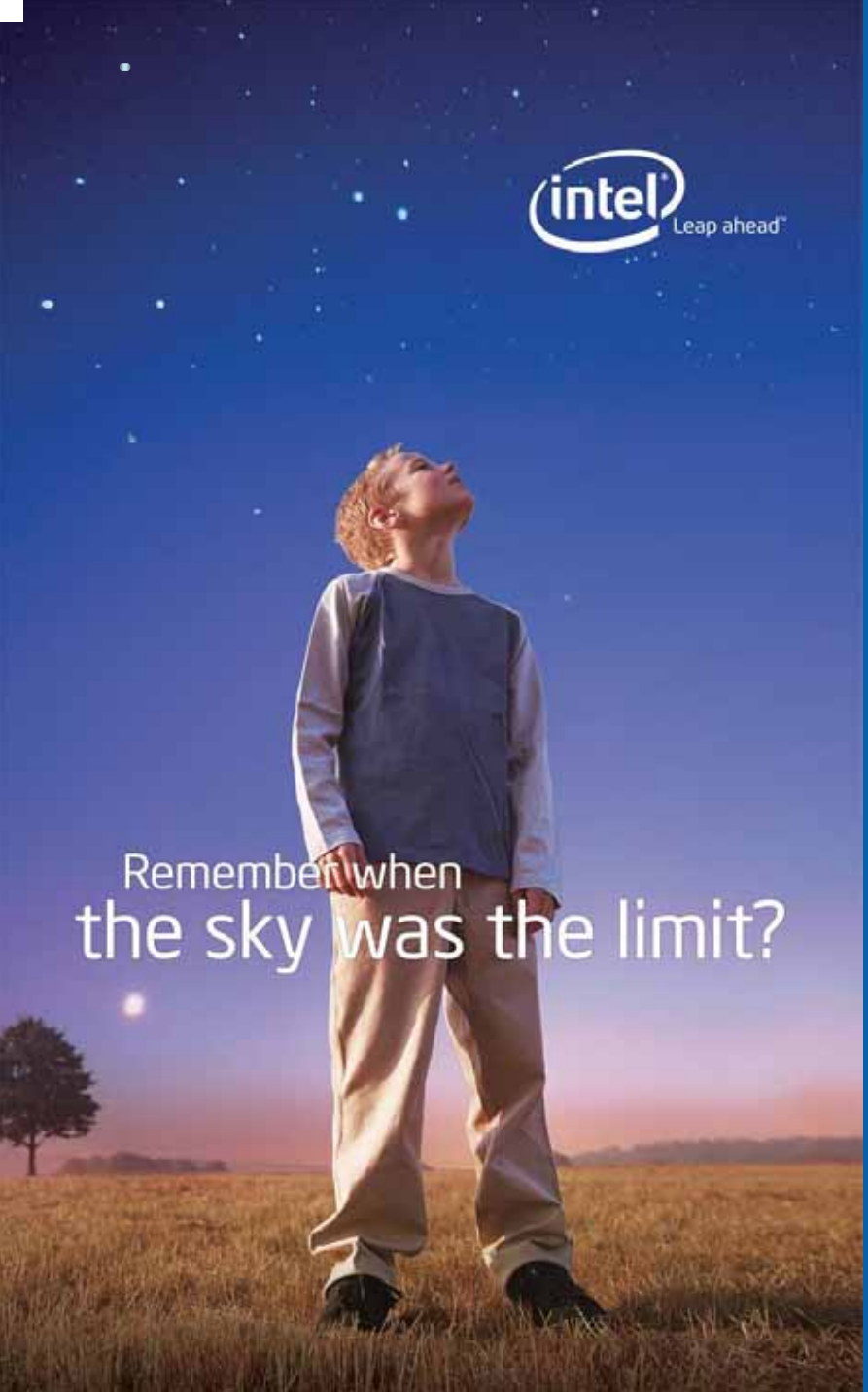


マルチコア対応アプリケーション開発における3つの課題

インテル株式会社
ソフトウエア製品部
菅原　清文

intel Software

本日の内容

インテルのソフトウエア開発製品への取り組み

インテルによるソリューション

- インテル® エクステンデッド・メモリー 64 テクノロジー
- デュアルコア
- ソフトウエア開発製品

マルチコア対応アプリケーション開発における3つの課題

インテルのソフトウエア製品開発部門

intel Software

なぜインテル® ソフトウエア開発製品が有効なのか？

パフォーマンス

- インテル® アーキテクチャ・プラットフォーム上で動作するパフォーマンスの高いソフトウエアの開発をサポート
- プラットフォームが出荷される前に、プラットフォームのパフォーマンスを引き出すコードの開発をサポート

互換性

- 広く利用されている主要な開発環境との互換性をサポート
  - 例：インテル® コンパイラは、一般的な gcc および Microsoft* Visual C++ のオプションを実装しており、GNU* gcc/g++ および Microsoft Visual .NET 製品と完全な相互運用性をサポート。同様に Etnus Totalview*、GNU GDB、とインテル® IDB デバッガは相互運用可能

サポート

- インテル® プレミア・サポートとインテル® ソフトウエア・カレッジにより経験豊富なサポートとツールの高度な利用法を提供

intel Software

常に最新のプラットフォームに対応

最新のプロセッサ技術動向にいち早く対応
マルチコアと 64 ビットに向けたソフトウェアの迅速かつシームレスな移行

インテルのソフトウェア最適化における優位性:

- インテルは 1980 年代からツールを開発
- 20 種類以上のツールとライブラリを提供
- 完全に IA（インテル® アーキテクチャ）に向けて構築
- ソフトウェア・パフォーマンスの大幅な向上による、IA プラットフォームの進化に貢献

主要企業・機関による採用:

NASA	IBM DB2	Oracle
Pixar	EA	MySQL
Adobe	Boeing	Shell
Morgan Stanley	Google	Raytheon

intel Software

インテルによるソリューション

インテル® エクステンデッド・メモリー 64 テクノロジー

デュアルコア

ソフトウエア開発製品

- スレッド化
- インテルのソフトウエア・ツールはデュアル コア・プロセッサーの優位性を引き出す
  - 並列化
  - パフォーマンス
  - 整合性

インテルはハードウェアを最大限に利用するためにソフトウェアをサポート

intel Software

インテル® エクステンデッド・メモリー 64 テクノロジー 概要

- 64 ビットの仮想アドレス空間をサポート
- フラットな仮想アドレス空間を提供
- IP相対アドレッシング・モード
- 8 つの汎用レジスターを追加
- 8 つの XMM レジスターを追加（128ビット）
- 64 ビット幅の汎用レジスター、命令ポインターおよび命令セット

マルチコアの可能性

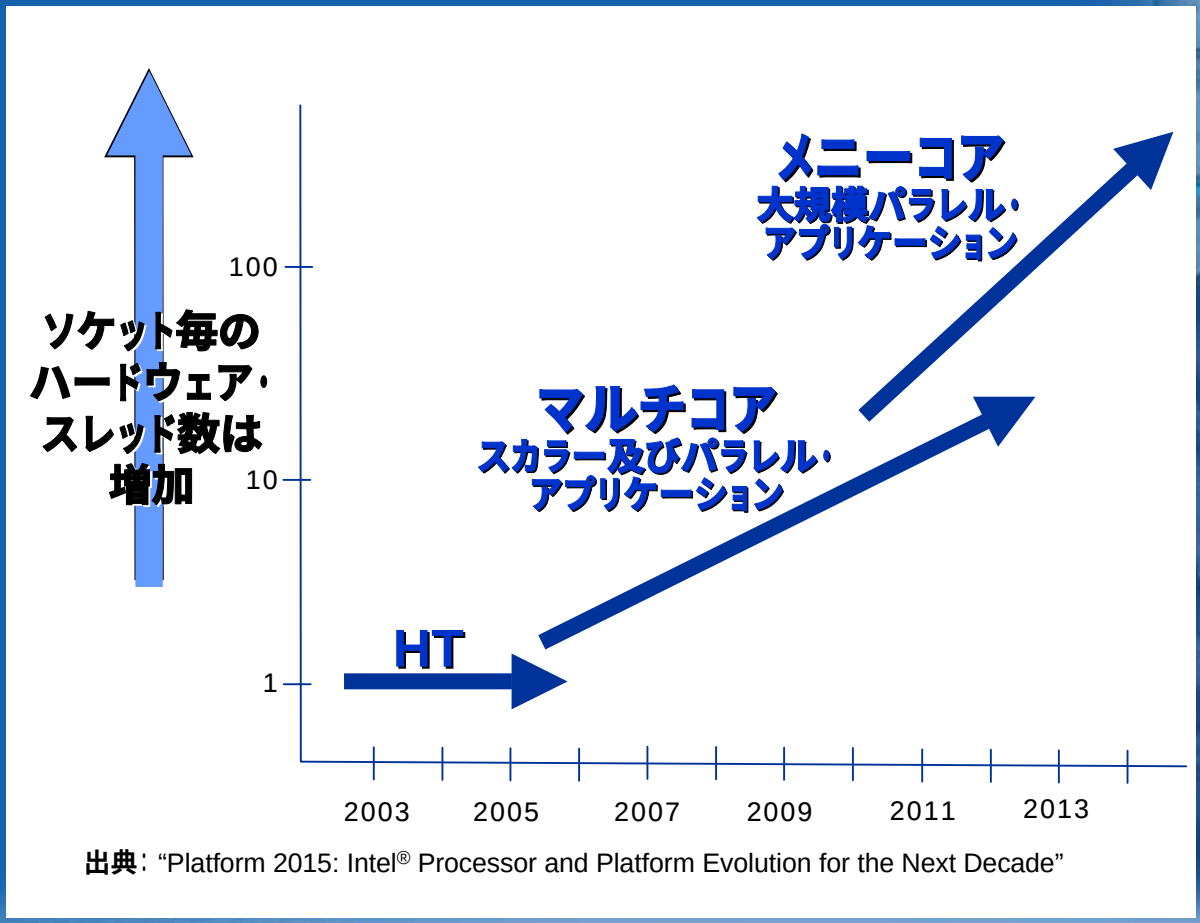
各社から発表される多くの新製品がマルチコア・プロセッサーを搭載

それらはソフトウエア開発者に挑戦と可能性という刺激的な世界をもたらす

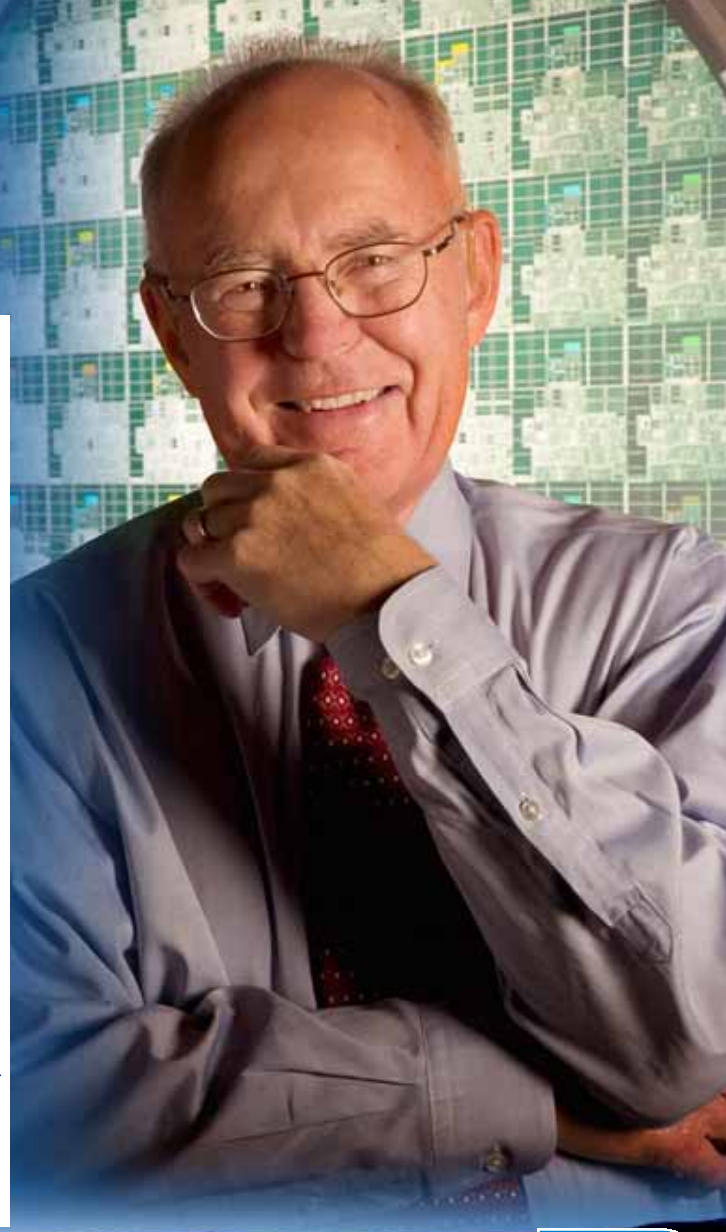
そこにはソフトウェア開発者が注目しなければいけない3つの課題があり、ソフトウェア開発ツールが並列化プログラミングという開発者の挑戦を手助けする

このセッションではこれらの課題について触れる

今後10年のインテル® プロセッサーとプラットフォームの発展



出典：“Platform 2015: Intel® Processor and Platform Evolution for the Next Decade”



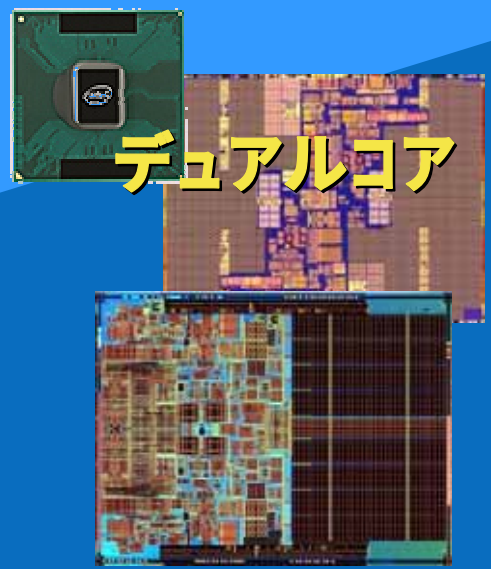
マルチコアのリーダーシップ：ラップトップ、デスクトップ、サーバー …

2004

Pentium® 4
Itanium® 2

2005-2006

Pentium D
Core™ Duo
Xeon® 5100
Itanium® 2 9100



2007+

Clovertown
Tukwila

ハイパフォーマンス ＋
低消費電力 ＋
小さなフォームファクター

intel Software

マルチコアへのマーケットの移行
インテル・プロセッサーの出荷予想**

	2005	2006年末の比率	2007年末の比率
高性能デスクトップ***	出荷	>70%	>90%
高性能モバイル***	出荷	>70%	>90%
サーバー	出荷	>85%	~100%

デスクトップ クライアント

モバイル クライアント

サーバー及び ワークステーション

*** モバイル及びデスクトップ Pentium

**本資料に記載されているすべての製品、日付、および数値は、現在の予想に基づくものであり、計画以外の目的ではご利用になれません。予告なく仕様が変更される場合があります。

出荷される大部分のインテル・プロセッサーは2006年末までにはデュアルコア版となる

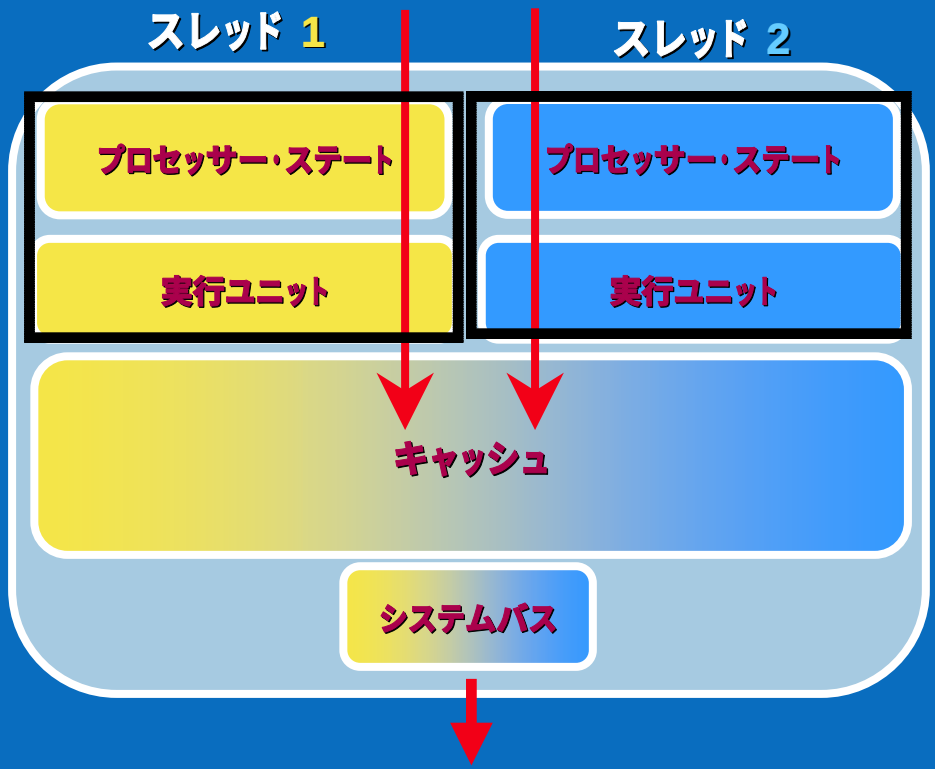
intel Software

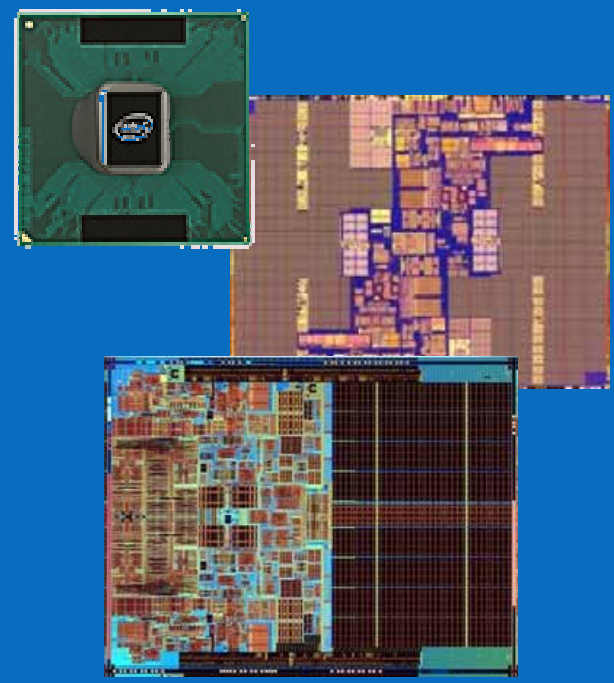
マルチコア

マルチコアは、” ソフトウェア開発において、オブジェクト指向による変革以来の最も大きな波” を作り出す

- Herb Sutter, Microsoft Senior Software Architect

デュアルコアの例





intel Software

マルチコアの優位性

マルチコア・プロセッサーを搭載するシステムは、同時に多くの処理を行い、より短い時間で処理を終えることができる

これを可能にするのがマルチタスクであり、コンピューターは複数のアプリケーションとOSのタスクを処理でる。マルチコアはより多くの処理を可能にする。

問い：マルチコアを1つのアプリケーションで利用できるか？

答え：はい、利用できます。

では、どのように並列処理を導入するか紹介します

並列処理の導入手順

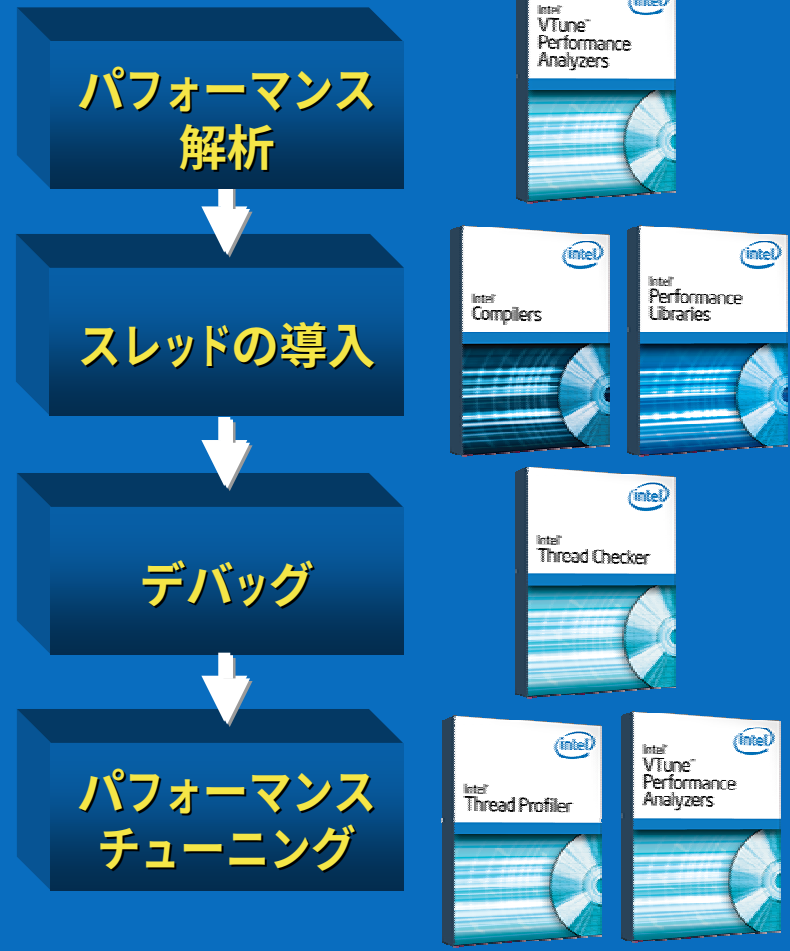
インテル® VTune™ アナライザー

インテル® C++ コンパイラー
インテル® Fortran コンパイラー
による OpenMP* サポート

インテル® パフォーマンス・ライブラリー
(MKL、IPP等)

インテル® スレッド・チェッカー

インテル® スレッド・プロファイラー、
インテル® VTune™ アナライザー



intel Software

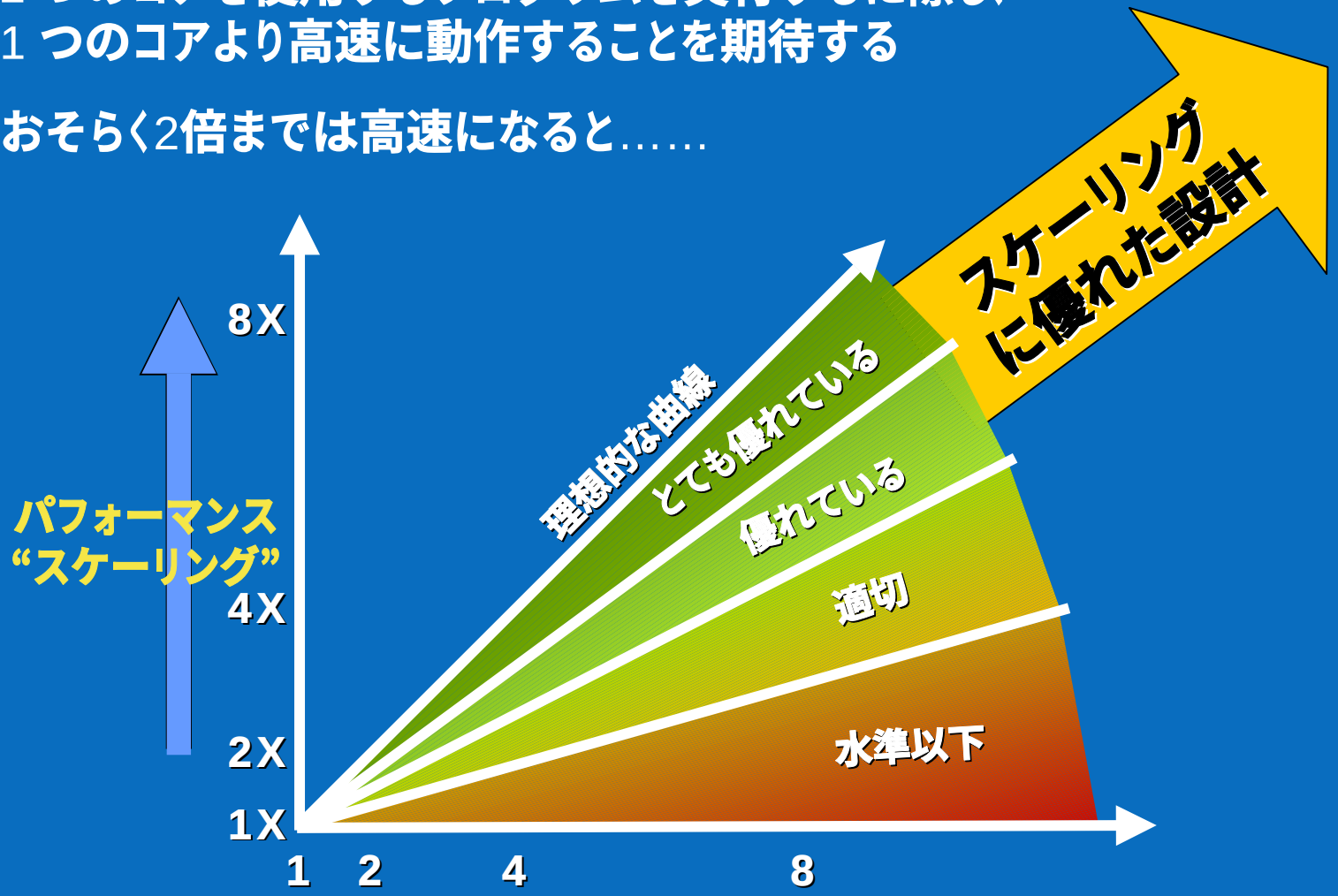
並列化プログラミングにおいて直面する 3 つの課題

- スケーラビリティー
- 正当性
- 容易な
  プログラミングと保守

スケーラビリティー

2 つのコアを使用するプログラムを実行するに際し、
1 つのコアより高速に動作することを期待する

おそらく2倍までは高速になると……



intel Software

スケーラビリティー

2 つのコアを使用するプログラムを実行するに際し、1 つのコアより高速に動作することを期待する

おそらく2倍までは高速になると……

重要なこと:

- パフォーマンスの解析
- 容易な実装
- 容易な保守
- 標準化がキー
  - MPI – 非常にスケーラブル
  - OpenMP* – スケーラブルで容易な保守性

インテル® ソフトウェア開発製品はこれらを容易に行うことを支援

intel Software

Win32 API を使用したスレッド化されたコード

レンダリングを行うタスクに加え、Win32 API ではスレッドの生成、負荷分散、同期、終了などを管理しなければいけない

```
CRITICAL_SECTION  lock;
HANDLE threads[threadsCount];


for (total = 0, row = 0; row <= scene.window.height; row += blockSize[X])
   for (col = 0; col <= scene.window.width; col += blockSize[Y], total++) {
      PRectangle *pProblem;
      pProblem = (PRectangle*)allocator.Alloc(sizeof(PRectangle));
      assert(pProblem != NULL);
      pProblem->left = col, pProblem->bottom = row;
      pProblem->width =
         (col + blockSize[X] <=
          scene.window.width) ? blockSize[X] :
                    scene.window.width - col + 1;
      pProblem->height =  (row + blockSize[Y] <=
                                  scene.window.height) ? blockSize[X] :
         scene.window.height - row + 1;
      Queue_Push(&problems, pProblem);
   }


/* initialize critical section object */
InitializeCriticalSection (&lock);
/* create threads */
for (int i = 0; i < threadsCount; i++)
threads[i] = CreateThread(0, 0, Render, NULL, 0, NULL);
/* wait for threads to complete */
WaitForMultipleObjects(threadsCount, threads, TRUE, INFINITE);
```

```
DWORD WINAPI Render(void *args) {
   for (;;) {
      /* lock Critical section object */
      EnterCriticalSection(&lock);

      /* fetch next problem from queue */
      PRectangle *pProblem = (PRectangle*)Queue_Pop(&problems);

      /* terminate loop if there are no problems left in queue */
      if (pProblem == NULL) {
            LeaveCriticalSection(&lock);
            break;
      }

      /* unlock Critical section object */
      LeaveCriticalSection(&lock);

      /* render part of image */
      Scene_Render(&scene, image.pPixels, pProblem);
      allocator.Dealloc(pProblem);
   }

   /* finish thread */
  return 0;
}
```


intel Software

OpenMP* によるコード

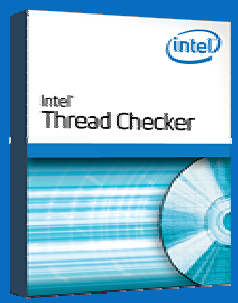
```
for (total = 0, row = 0; row <= scene.window.height; row += blockSize[X])
    for (col = 0; col <= scene.window.width; col += blockSize[Y], total++) {
        PRectangle *pProblem;
        pProblem = (PRectangle*)allocator.Alloc(sizeof(PRectangle));
        assert(pProblem != NULL);
        pProblem->left = col, pProblem->bottom = row;
        pProblem->width =
            (col + blockSize[X] <=
             scene.window.width) ? blockSize[X] :
                 scene.window.width - col + 1;
        pProblem->height =
            (row + blockSize[Y] <=
             scene.window.height) ? blockSize[X] :
            scene.window.height - row + 1;

        Queue_Push(&problems, pProblem);
    }
#pragma omp parallel default(shared)
{
#pragma omp for schedule(dynamic)
    for (int processed = 0; processed < total; processed++) {
        PRectangle *pProblem;
#pragma omp critical
        {
            /* fetch next problem from queue */
            pProblem = (PRectangle *) Queue_Pop (&problems);
        }
        /* render part of image */
        Scene_Render (&scene, image.pPixels, pProblem);
        allocator.Dealloc (pProblem);
    }
}
```

OpenMP を使用するとこにより、スレッドの管理等を行うことなく容易に並列化ができる

OpenMP は明示的なスレッドのラッパーやスレッド化する関数を作成する煩わしさをなくす

正当性

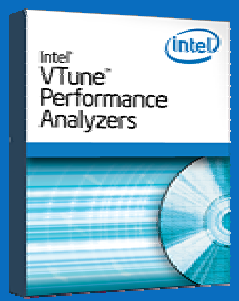
アプリケーションをデバッグする際、
アプリケーションを展開する前に問題を検出



潜在するスレッディングの問題をピンポイントで検出

インテル® スレッド・チェッカー

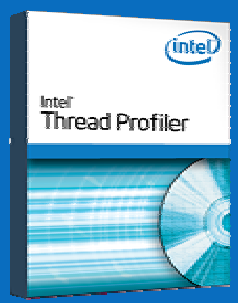
特許を取得した高度なエラー検出エンジンは、
潜在するデータ競合やデッドロックを明らかにします。



スレッド化されたコード内部の正確な性能を測定

インテル® VTune™ パフォーマンス・アナライザー:

アプリケーション/ロック・レベル

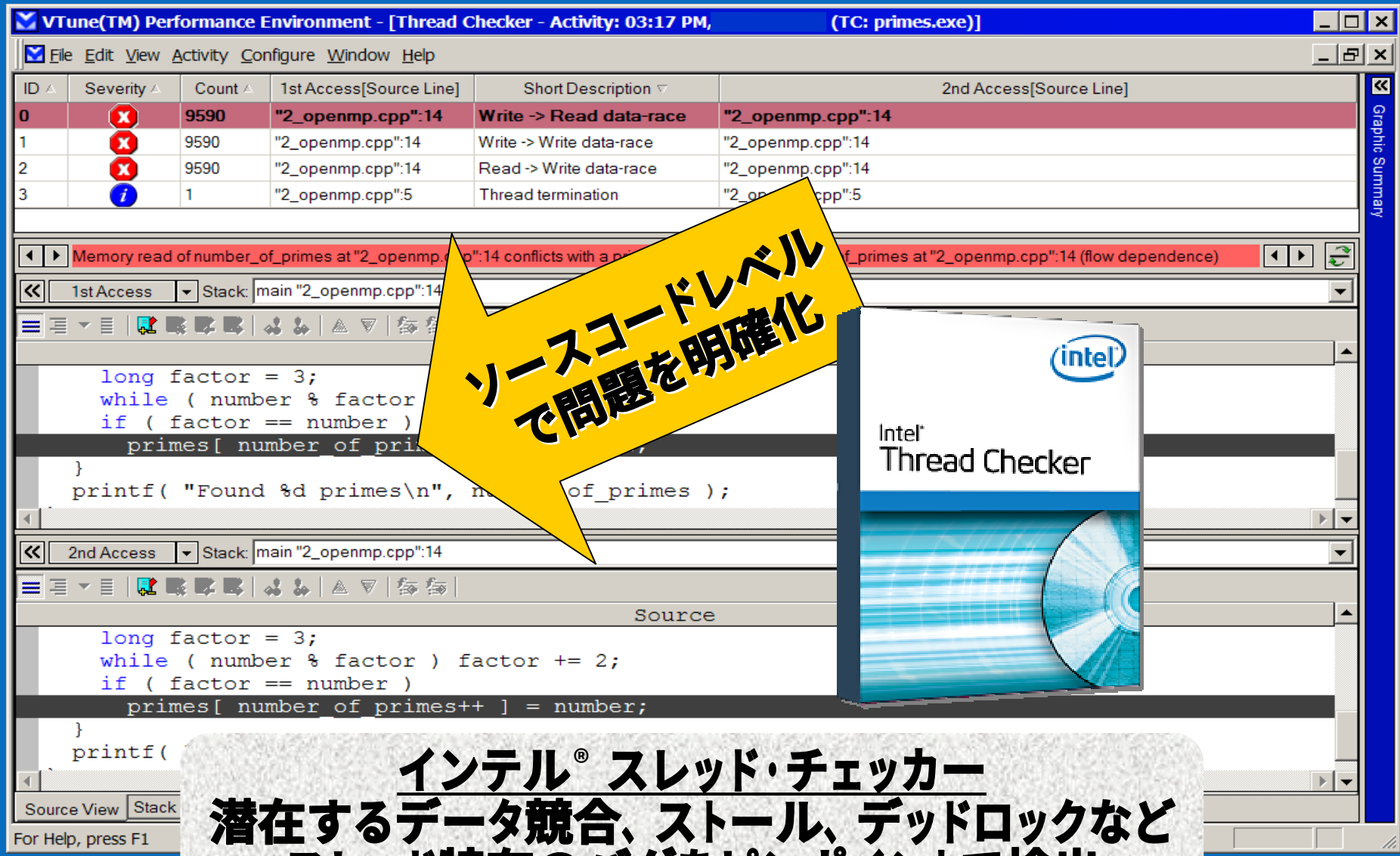


インテル® スレッド・プロファイラー:

アプリケーション/システム レベル

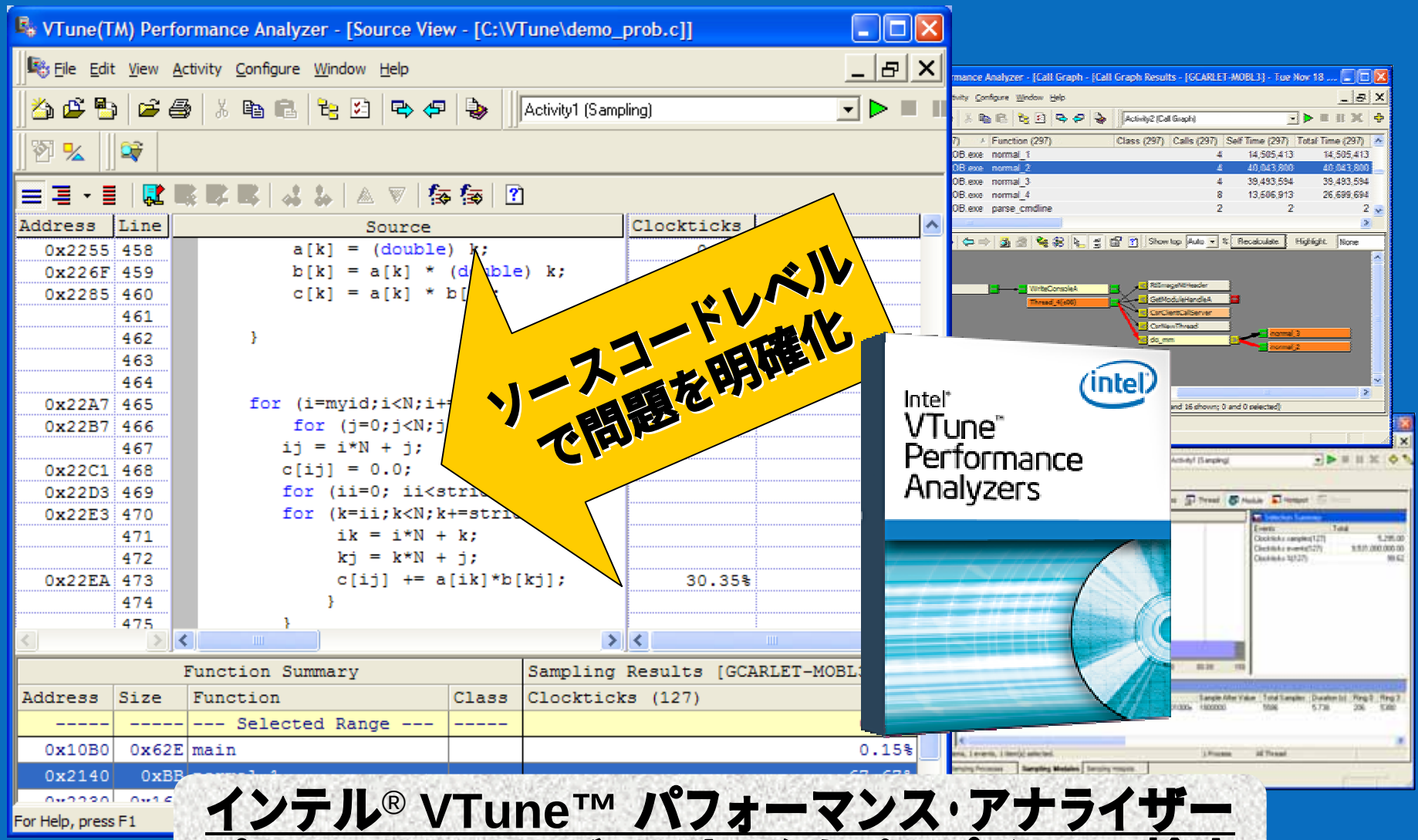
容易な正当性の確認を支援

intel Software

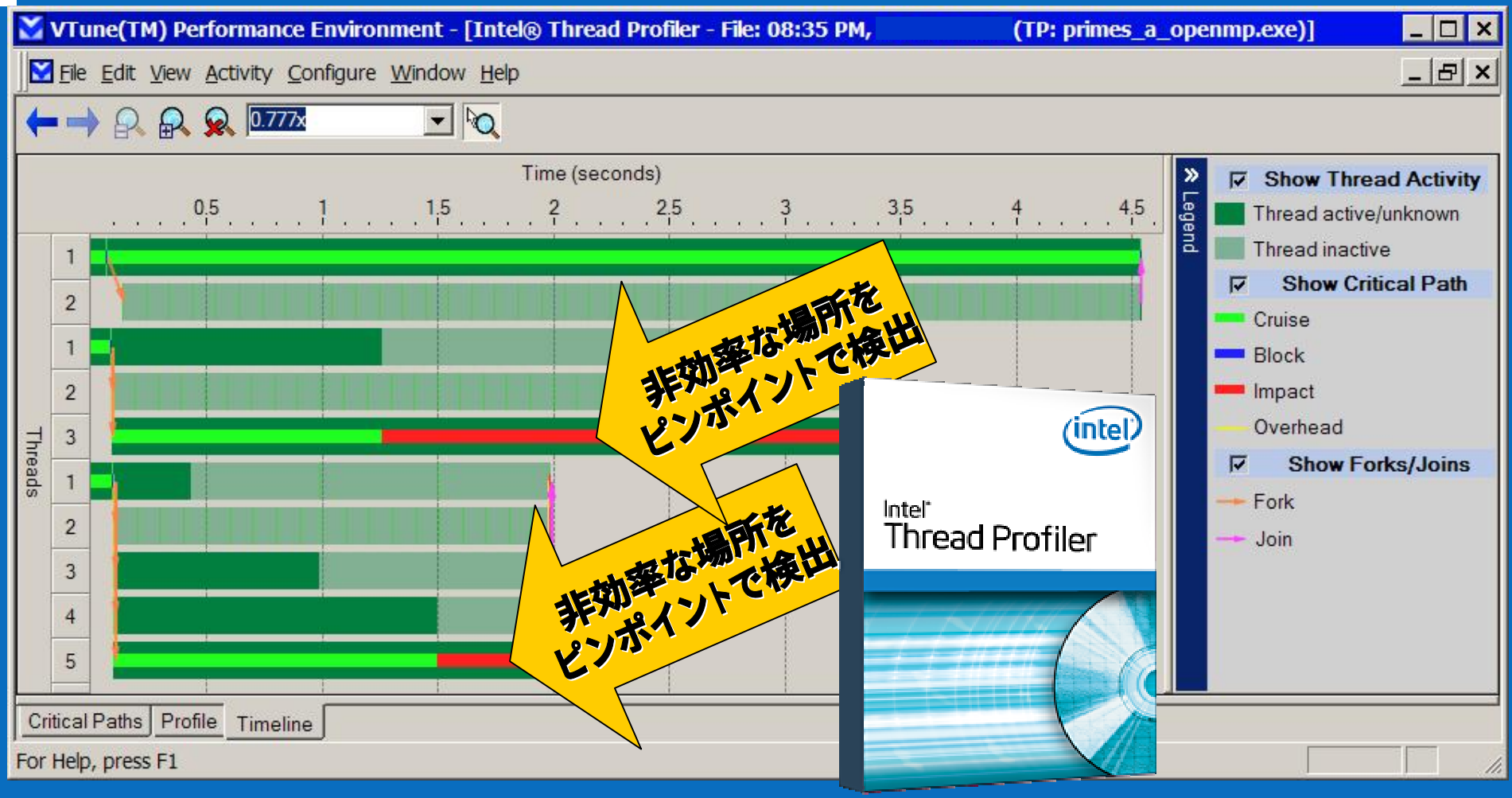


インテル® スレッド・チェッカー
潜在するデータ競合、ストール、デッドロックなど スレッド特有のバグをピンポイントで検出

intel Software



インテル® VTune™ パフォーマンス・アナライザー
パフォーマンスのボトルネックをピンポイントで検出



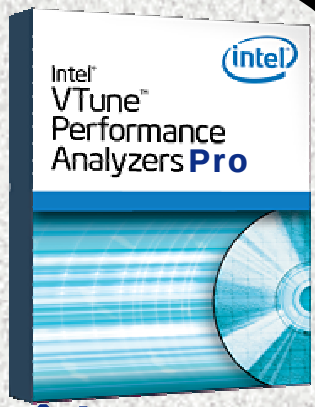
インテル® スレッド・プロファイラー
非効率なスレッディングの問題をピンポイントで検出

intel Software

正当性

アプリケーションをデバッグする際、
アプリケーションを展開する前に問題を検出

まもなく
登場

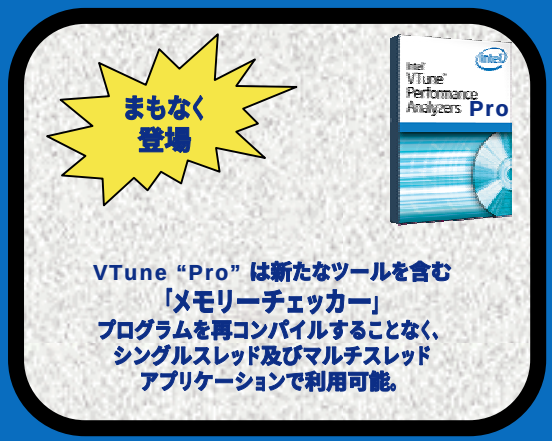


VTune “Pro” は新たなツールを含む
「メモリーチェッカー」
プログラムを再コンパイルすることなく、
シングルスレッド及びマルチスレッド
アプリケーションで利用可能です。

intel Software

正当性

アプリケーションをデバッグする際、
アプリケーションを展開する前に問題を検出



- 再コンパイルの必要なし
- ソースコードは必要なし
- スレッドは正確にハンドルされる
- スレッドは効率的にハンドルされる
- バウンズ・チェック
- ヒープポインターのチェック
- リークのスナップショット
- ダングリング・ポインターへの読み書きを検出
- 不正な “free()” の利用を検出

容易なプログラミングと保守

並列コードをシンプルに記述
将来に亘って保守できるコードを記述

標準化された MPI と OpenMP* は重要

インテルではライブラリー（インテル® MKL とインテル® IPP）における並列化の実装に使用しています。

インテルではクラスター・ライブラリー（インテル® クラスター MKL）の実装に MPI を使用しています。

インテルの解析ツールは MPI および OpenMP を活用し、
MPI と OpenMP に関して結果を残しています

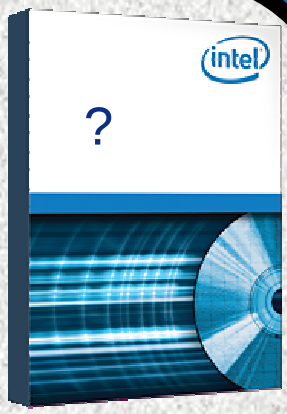
容易なプログラミングと保守を支援

intel Software

容易なプログラミングと保守

並列コードをシンプルに記述
将来に亘って保守できるコードを記述

まもなく
登場



スレッドではなくタスクのパターンをプログラムする

インテルの次世代プログラミング・モデル
C++ テンプレートを元にしたランタイム・ライブラリーは、
マルチスレッド・アプリケーションの記述を単純化

スケーラビリティーに優れた新しい
高度なプログラミング・パラダイム

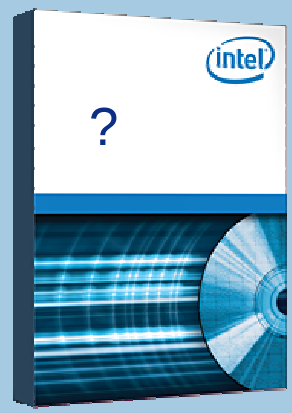
intel Software

新しいネイティブ・スレッド・コード

```
const int MINPATCH = 150;
const int DIVFACTOR = 2;

typedef struct work_queue_entry_s {
    patch pch;
    struct work_queue_entry_s *next;
} work_queue_entry_t;
work_queue_entry_t *work_queue_head = NULL;
work_queue_entry_t *work_queue_tail = NULL;

void generate_work (patch* pchin)
{
    int startx, stopx, starty, stopy;
    int xs,ys;

    startx=pchin->startx;
    stopx= pchin->stopx;
    starty=pchin->starty;
    stopy= pchin->stopy;

    if(((stopx-startx) >= MINPATCH) || ((stopy-starty) >= MINPATCH)) {
      int xpatchsize = (stopx-startx)/DIVFACTOR + 1;
      int ypatchsize = (stopy-starty)/DIVFACTOR + 1;
      for (ys=starty; ys<=stopy; ys+=ypatchsize)
      for (xs=startx; xs<=stopx; xs+=xpatchsize) {
        patch pch;
        pch.startx = xs;
        pch.starty = ys;
        pch.stopx  = MIN(xs+xpatchsize-1,stopx);
        pch.stopy  = MIN(ys+ypatchsize-1,stopy);

        generate_work (&pch);
      }
    } else {
      /* just trace this patch */
      work_queue_entry_t *q = (work_queue_entry_t *) malloc (sizeof (work_queue_entry_t));
      q->pch.starty = starty;
      q->pch.stopy = stopy;
      q->pch.startx = startx;
      q->pch.stopx = stopx;
      q->next = NULL;
if (work_queue_head == NULL) {
          work_queue_head = q;
      } else {
          work_queue_tail->next = q;
      }
      work_queue_tail = q;
    }
}

void generate_worklist (void)
{
    patch pch;
    pch.startx = startx;
    pch.stopx = stopx;
    pch.starty = starty;
    pch.stopy = stopy;
    generate_work (&pch);
}

bool schedule_thread_work (patch &pch)
{
    EnterCriticalSection (&MyMutex3);
    work_queue_entry_t *q = work_queue_head;
    if (q != NULL) {
        pch = q->pch;
        work_queue_head = work_queue_head->next;
    }
    LeaveCriticalSection (&MyMutex3);
    return (q != NULL);
}

generate_worklist ();
```

新しいコード…

```
#include "tbb/ParallelFor.h"
#include "tbb/BlockedRange2D.h"

ParallelFor (TBB::BlockedRange2D<int> (starty, stopy + 1,
grain_size, startx, stopx + 1, grain_size), parallel_task ());
```



intel Software

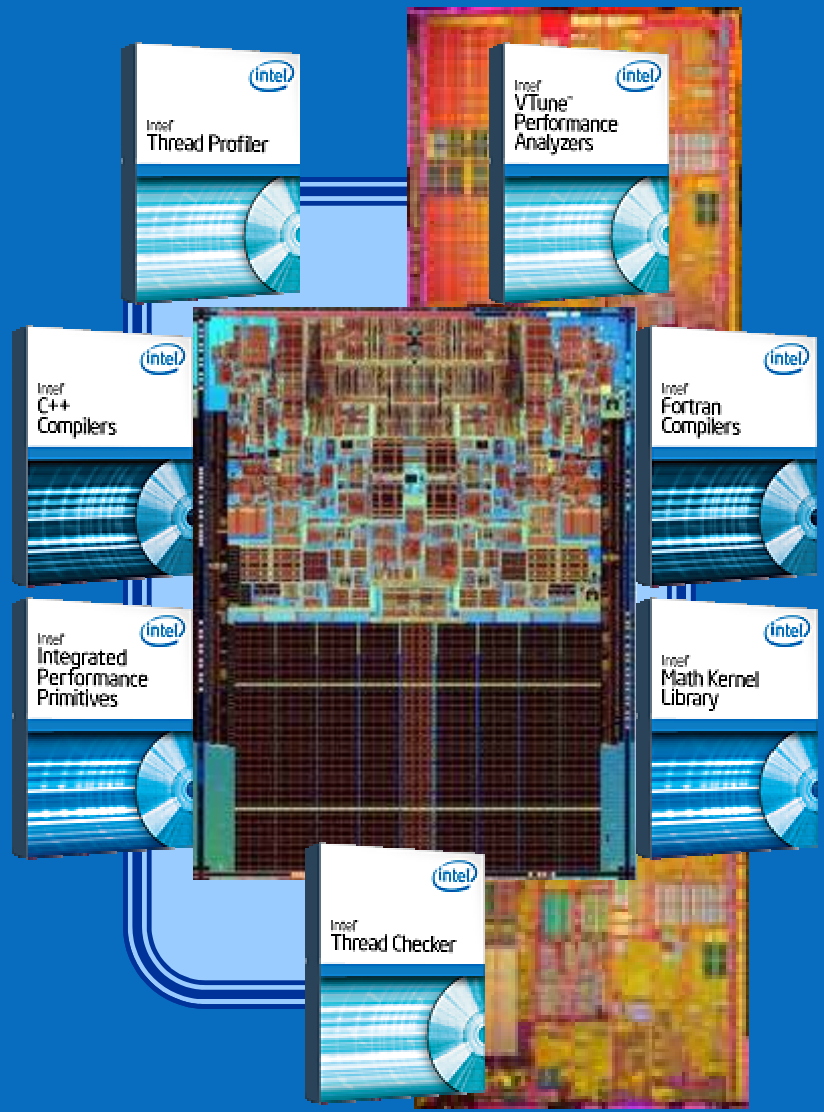
まとめ

インテル:

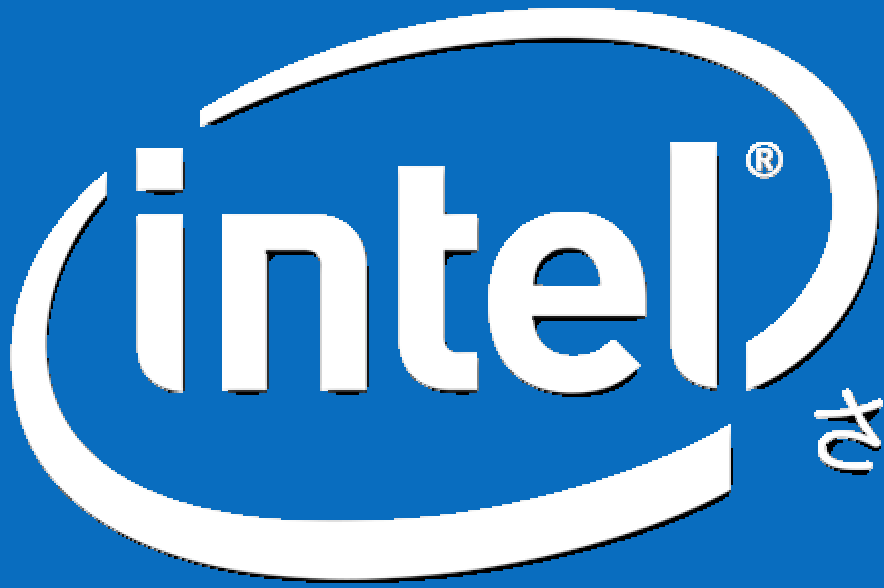
プロセッサーを
マルチコア化され

そして

ソフトウェア開発ツールは、
3つの重要な課題に注目



intel Software



さあ、その先へ。™