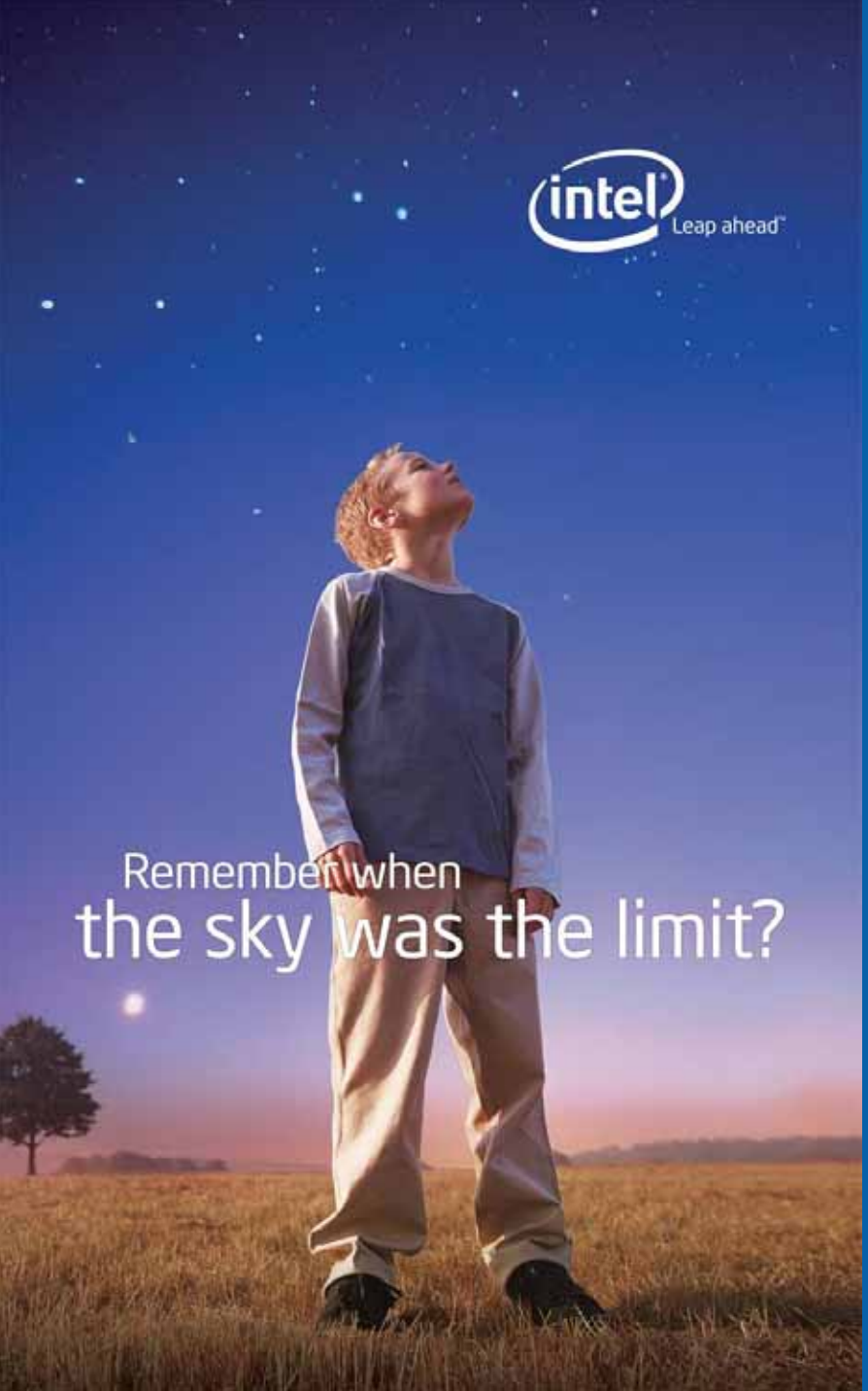


スレッド化の概念と導入

Aug 3 ‘06

intel
Software

内容

1. 並列処理アーキテクチャーの進化を理解する
2. アーキテクチャーのスレッド化とソフトウェア開発との関係を示す
3. 時間のかかる領域をスレッド化するために必要な作業を素早くプロトタイプ作成し、計画できるようにする

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

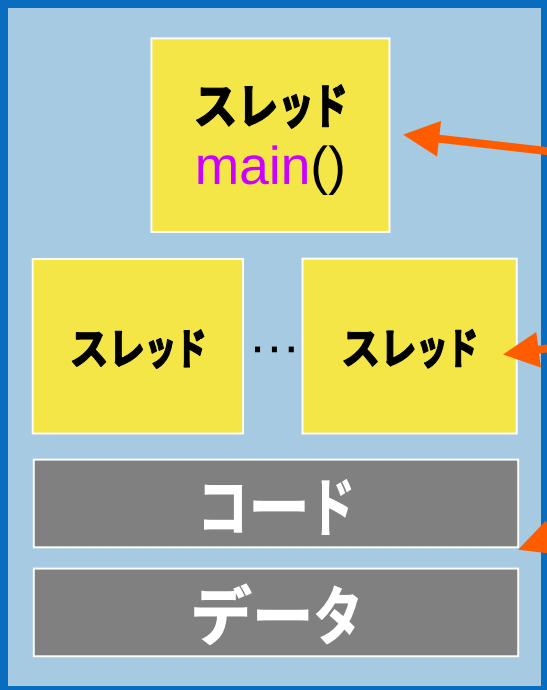
並列処理とは?

同時に 2 つ以上のプロセスまたはスレッドを実行すること

マルチスレッド・アーキテクチャーのための並列処理

- マルチプロセス
  - プロセス間通信 (IPC) による通信
- シングルプロセス、マルチスレッド
  - 共有メモリーによる通信

スレッドとプロセス



現在のオペレーティング・システムではプログラムをプロセスとしてロードする

プロセスはエントリーポイントでスレッドとして実行を開始する

スレッドはプロセス内で他のスレッドを作成できる

プロセス内のすべてのスレッドはコードとデータ領域を共有する

スレッドは 2 つのスケジュール状態（アクティブ、インアクティブ）を持つ

スレッド

長所

- パフォーマンスが向上し、リソースの使用率が改善される
  - シングル・プロセッサー・システムにおいてもレイテンシーの隠蔽とスループットの向上が見込める
- 共有メモリーによる IPC はより効率的

短所

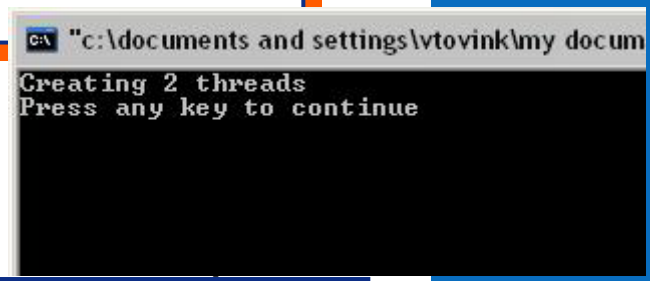
- アプリケーションがより複雑になる
- デバッグ (データの競合、デッドロック、その他) が困難

Hello World

```
DWORD WINAPI HelloFunc (LPVOID)
{
        cout << "Hello Thread¥n";
        return 0;
}
```

```
{
        HANDLE hThread[numThreads];

        for (int i = 0; i < numThreads; i++)
            hThread[i] = CreateThread (NULL, 0,
```




```
for (int i = 0; i < numThreads; i++)
   hThread[i] = CreateThread (NULL, 0, HelloFunc,
                                      NULL, 0, NULL );

// Clean up thread handles
for (int i = 0; i < numThreads; i++)
        CloseHandle (hThread[i]);

return 0;
}
```

Hello World (変更後)

```
for (int i = 0; i < numThreads; i++)
    hThread[i] = CreateThread (NULL, 0, HelloFunc,
                                       NULL, 0, NULL );
getchar();

// Clean up thread handles
for (int i = 0; i < numThreads; i++)
       CloseHandle (hThread[i]);

return 0;
```

サンプル･コードの変更

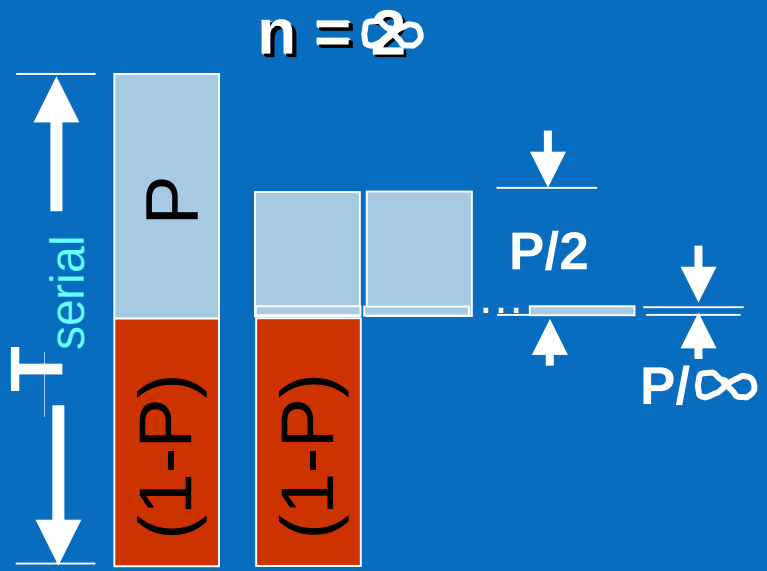
- “Hello World Thread <スレッド番号>” メッセージに変更するには？

```
Hello World Thread #1
Hello World Thread #0
.. .. .. ..
```

アムダールの法則

並列化のスピードアップ（スケーリング）の上限を説明

オーバーヘッドの影響を考慮する際に役立つ



$$T_{parallel} = \{(1-P) + P/n + O\} T_{serial}$$

0.5 + 0.05

n = プロセッサーの数

スケーリング = $T_{serial} / T_{parallel}$ 1.0/0.55 = 1.33

シリアルコードがスケーリングを制限する

並列プログラミング・モデル

機能分割

- タスクの並列処理
- 同じ問題の独立したタスク
  - UI タスク、UI 更新、生産と消費のパラダイム
  - 並列演算と次の表示タスク

データ分割

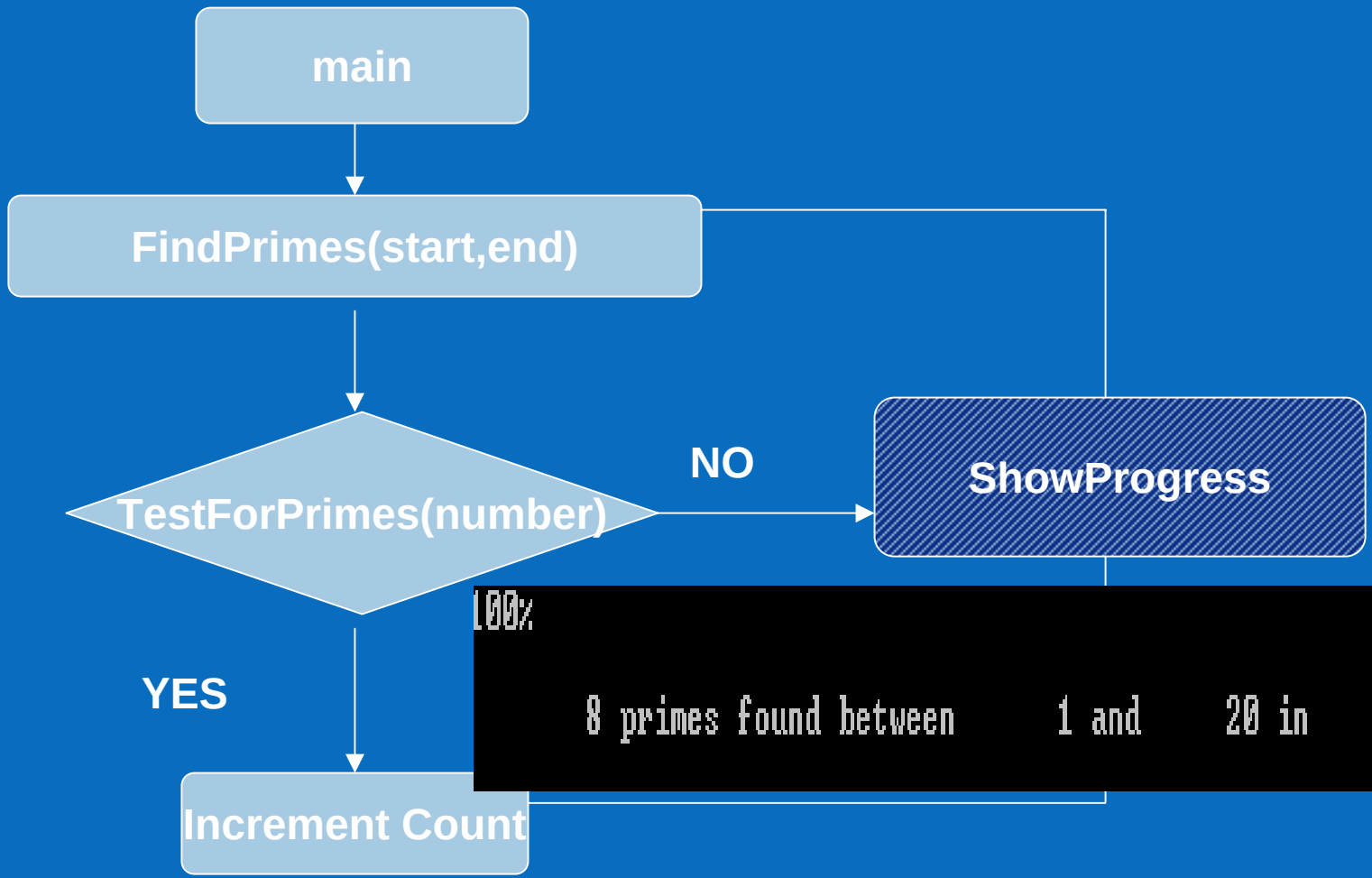
- 異なるデータで実行される同じ演算
- 例: 行列の乗算

アプリケーションをスレッド化する際によくある質問

- どこをスレッド化すればいいのか?
- スレッド化に必要な時間は?
- 必要な再設計の回数は?
- 選択した領域をスレッド化する価値はあるか?
- どの程度のスピードアップを期待すべきか?
- パフォーマンスは期待値を満たすことができるか?
- プロセッサーを追加すれば性能が向上するか?
- どのスレッドモデルを使用すべきか?

ケーススタディー

素数生成



ケーススタディー

素数生成

2

3 3

5 3,5

7 3,5,7

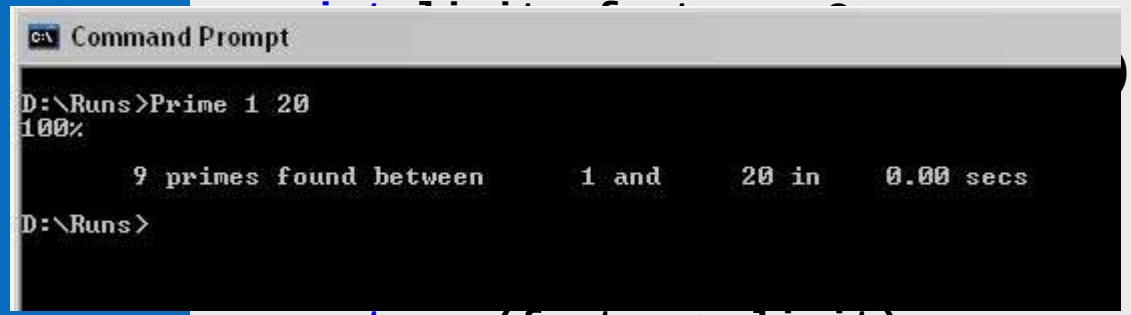
9 3

11 3,5,7,9,11

13 3,5,7,9,11,13

15 3

17 3,5,7,9,11,13




```
bool TestForPrime(int val)
{   // let's start checking from 3
                                          +0.5f);
    return (factor > limit);
}

void FindPrimes(int start, int end)
{
    // start is always odd
```

```
for( int i = start; i <= end; i+= 2 ){
    if( TestForPrime(i) )
        gPrimesFound++;
    ShowProgress(i, range);
}
```

一般的な開発サイクル

解析

- インテル® VTune™ パフォーマンス・アナライザー

設計 (スレッドの導入)

- インテル® パフォーマンス・ライブラリー: IPP および MKL
- OpenMP* (インテル® コンパイラー)
- 明示的なスレッディング (Win32*、Pthreads*)

正当性のデバッグ

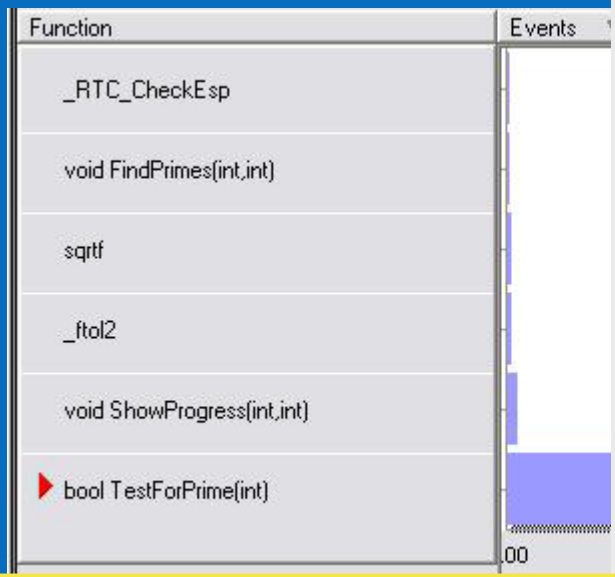
- インテル® スレッド・チェッカー
- インテル® デバッガー

パフォーマンスのチューニング

- スレッド・プロファイラー
- インテル® VTune™ パフォーマンス・アナライザー

解析 – サンプリング

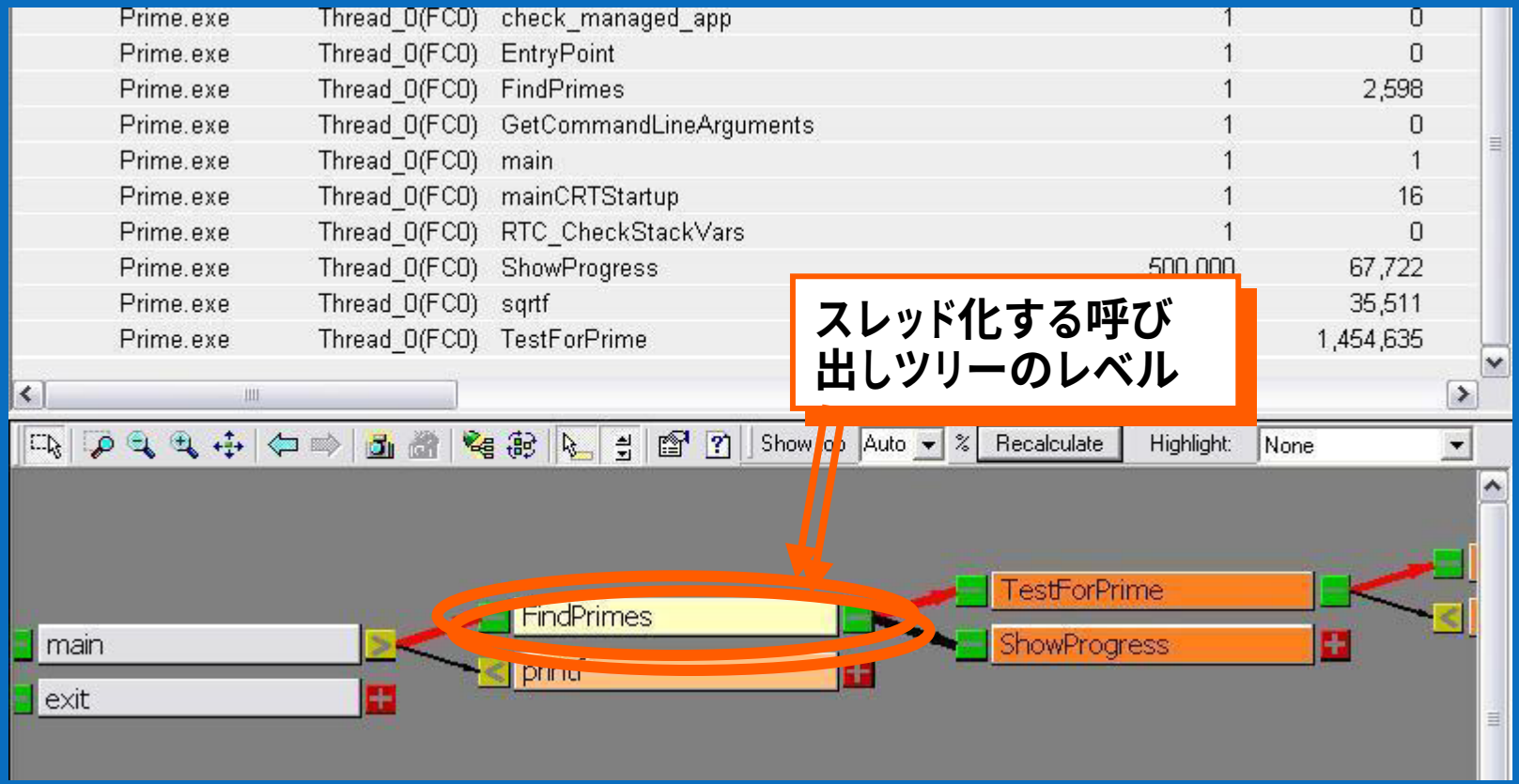
シングルスレッド “Prime”
VTune™ アナライザーの




```
void FindPrimes(int start, int end)
{
    // start is always odd
    int range = end - start + 1;
    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
            gPrimesFound++;
        ShowProgress(i, range);
    }
```

時間のかかる領域を特定する

解析 – コールグラフ



スレッド化する呼び出しツリーのレベルを特定する

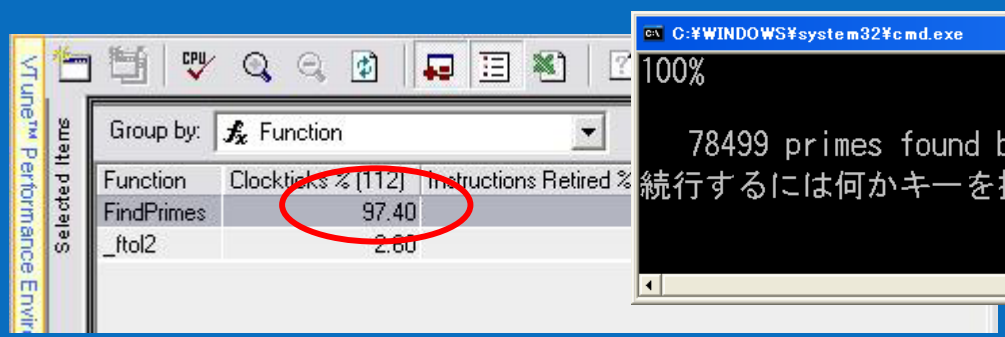
解析

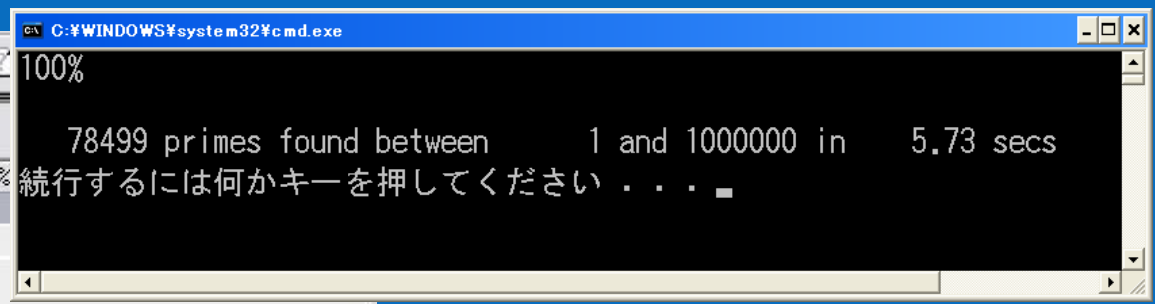
どこをスレッド化すればいいのか?

- FindPrimes()

選択した領域をスレッド化する価値はあるか?

- 依存関係が少ない
- データを並列化できる
- 実行時間の 95% 以上を占める





設計

予想されるメリットは?

最小限の作業で予想されるメリットを判断する方法は?

OpenMP* を使用した素早いプロトタイプ作成

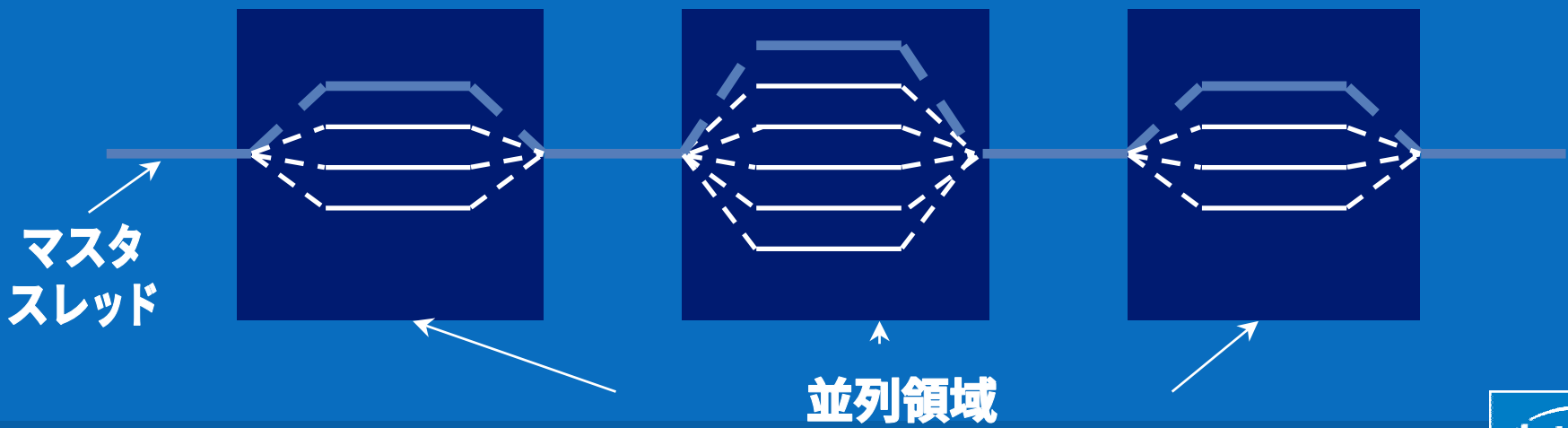
スレッド化に必要な時間は?

必要な再設計の回数は?

OpenMP*

Fork-join 型の並列処理:

- マスタスレッドは必要に応じて、スレッドのチームを生成する
- 並列処理は徐々に追加される。つまり、逐次処理プログラムは並列処理プログラムへ進化する



素数生成

設計

```
#pragma omp parallel for
    for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
            gPrim
    ShowProgress( i, end );
```

OpenMP*

for ループで定義する

この並列領域のスレッドをここで作成する

```
Intel(R) C++ Compiler 9.1.022  build environment for 32-bit applications
eSingle.exe 1 1000000
 50%

   78415 primes found between      1 and 1000000 in    1.74 secs

C:¥Documents and Settings¥ijkk¥デスクトップ>
```

設計

予想されるメリットは?

最小限の作業で予想されるメリットを判断する方法は?

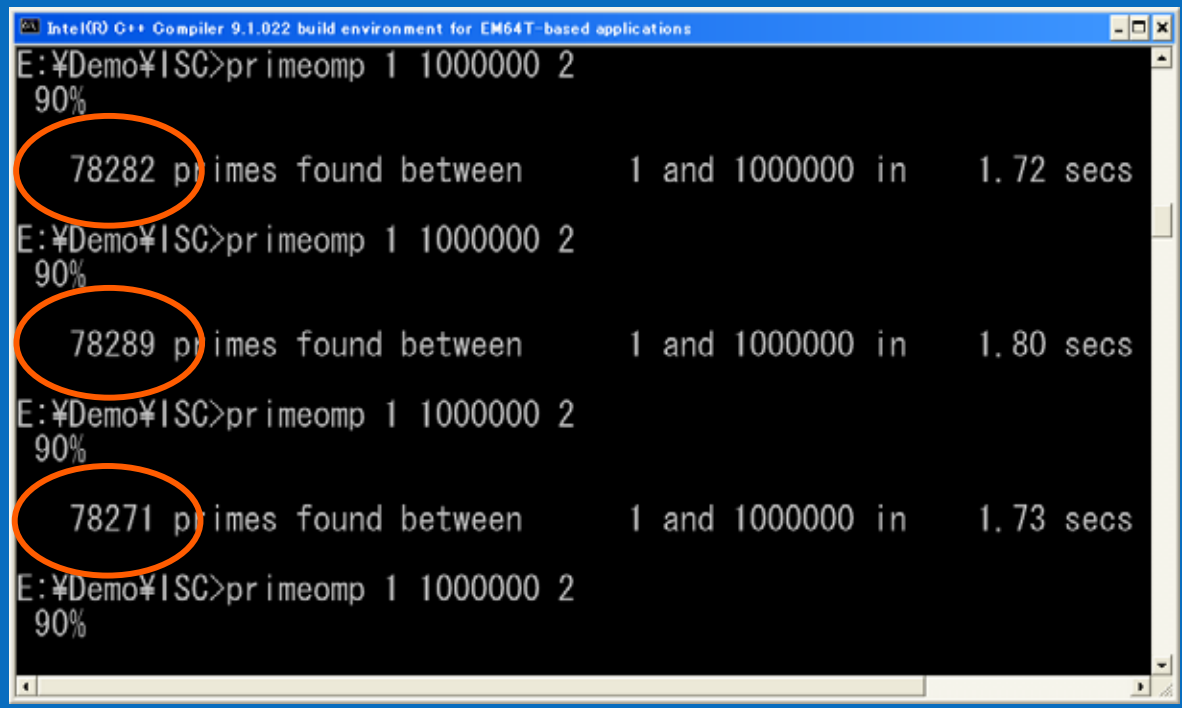
5.73/1.74 = 3.29X のスケーリング

スレッド化に必要な時間は?

必要な再設計の回数は?

可能な最良のスケーリングか?

正当性のデバッグ



毎回答えが同じであるか?

このスレッドの実装は正しいか?

データの競合

複数のスレッドで同じ変数が同時にアクセスされること

例:

- a=1、b=2 (a、b はグローバル変数) の場合を考える

スレッド1:

```
x = a + b ;
```

スレッド2:

```
b = 42 ;
```

x の値は?

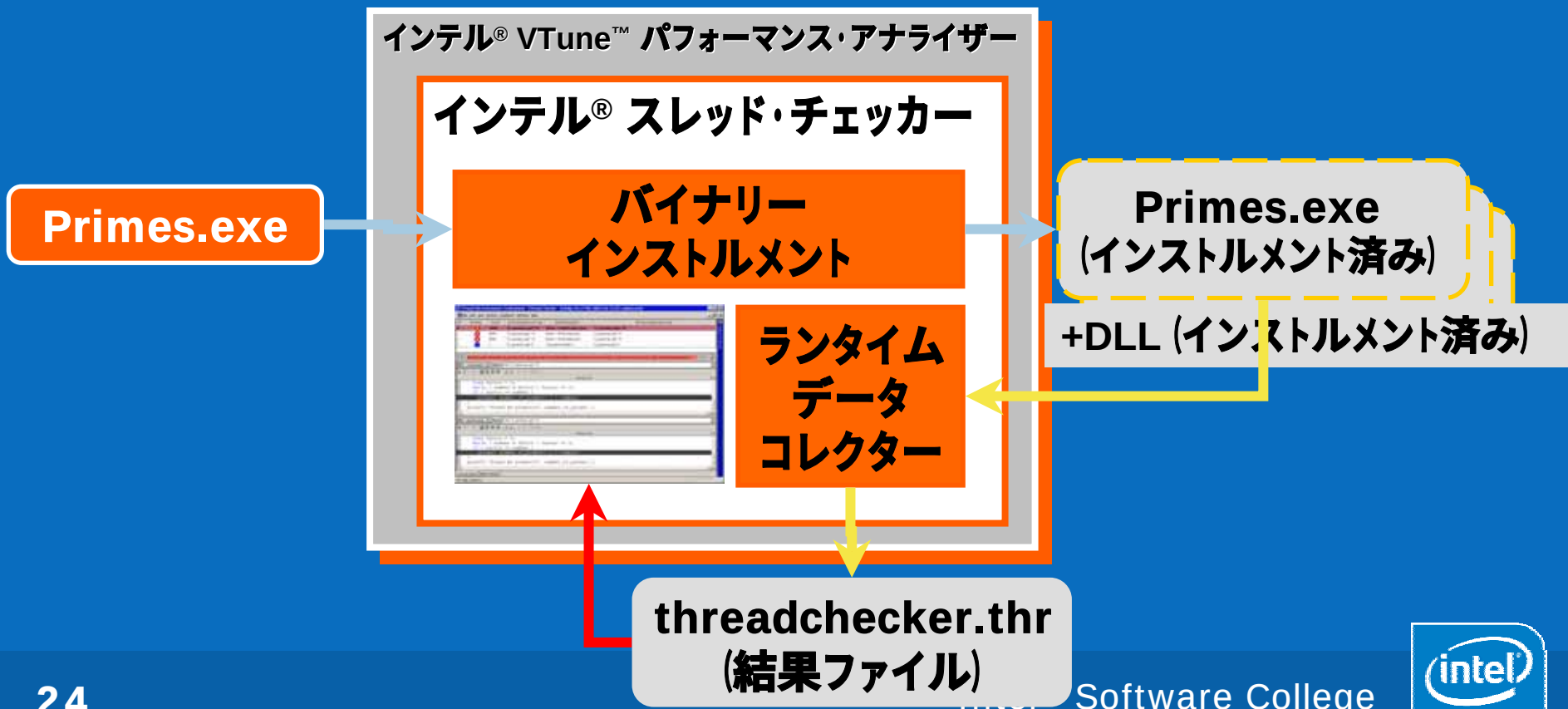
- **スレッド1 をスレッド2 の前に実行した場合**
- **スレッド1 をスレッド2 の後に実行した場合**
- **スレッド1 をスレッド2 と同時に実行した場合**
  - **どのような場合に発生するか?**

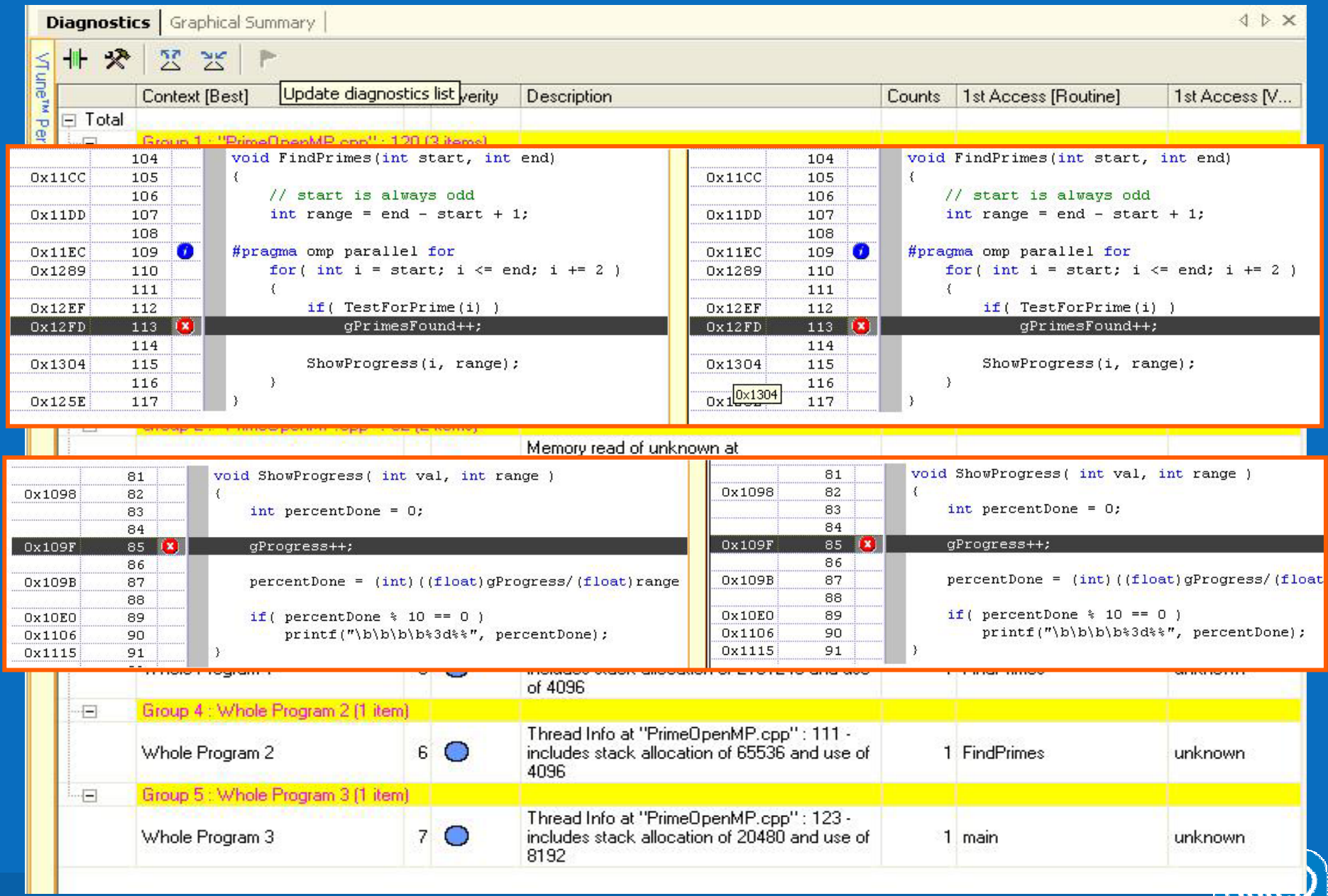
x = 1 + 2 = 3

x = 1 + 42 = 43

正当性のデバッグ

インテル® スレッド・チェッカーは、データの競合、ストール、およびデッドロックなどのスレッド化の問題を正確に検出する





正当性のデバッグ

```
#pragma omp parallel for
     for( int i = start; i <= end; i+= 2 ){
        if( TestForPrime(i) )
#pragma omp atomic
            gPrimesFound++;
        ShowProgress(i, range);
    }
#pragma omp critical
  {
    gProgress++;
    percentDone = (int)(gProgress/range *200.0f+0.5f)
  }
```

アトミック演算でインクリメントする

クリティカル・セクションを作成する

正当性のデバッグ

スレッド化に必要な時間は?

必要な再設計の回数は?

**スレッド・チェッカーで検出された依存関係は
2 つだけなので必要な作業は少ない**

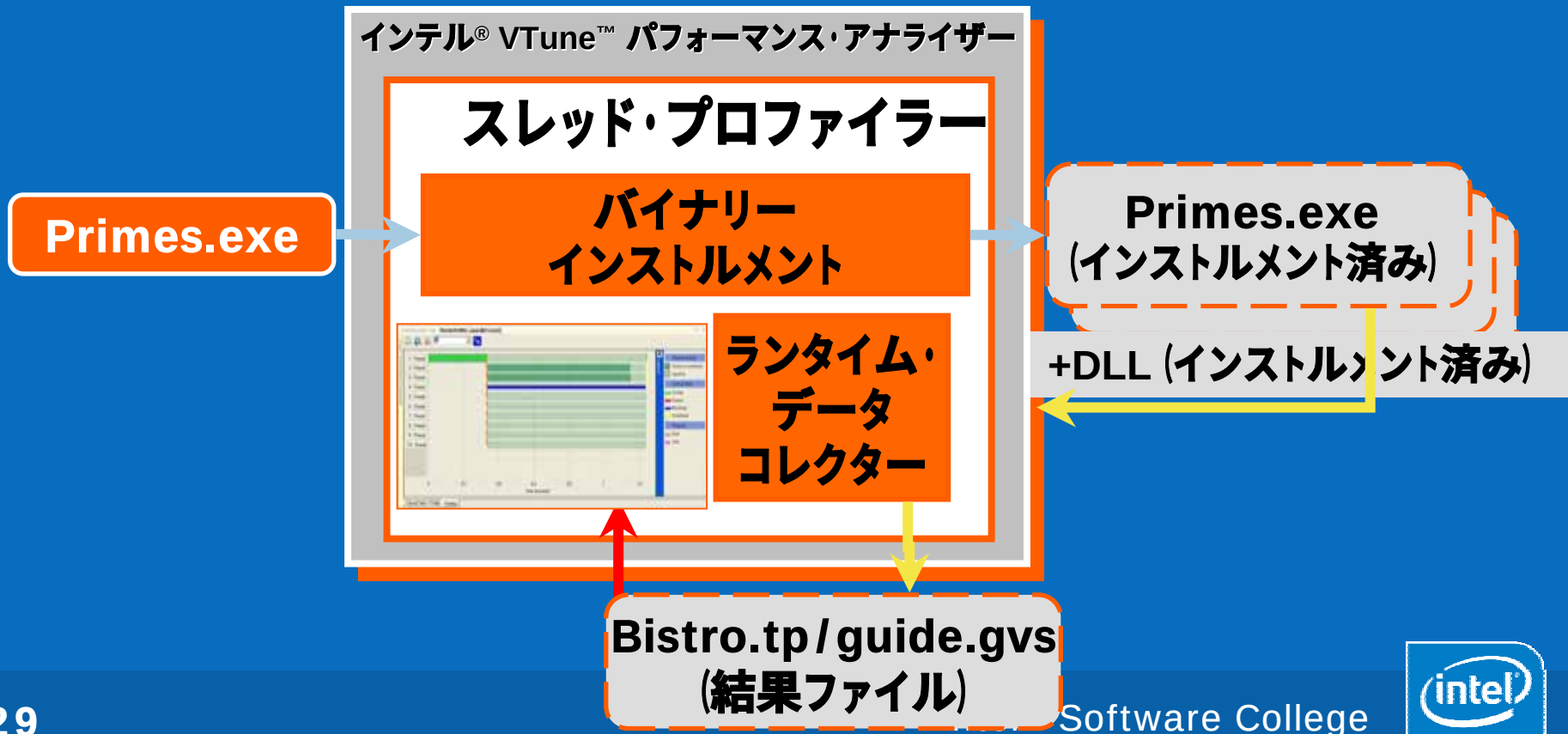
パフォーマンス

3.29X のパフォーマンスは最適か?

このアルゴリズムから期待できる最良のパフォーマンスか?

パフォーマンスのチューニング

スレッド・プロファイラーは、スレッド化したアプリケーションの
パフォーマンス・ボトルネックを正確に検出する

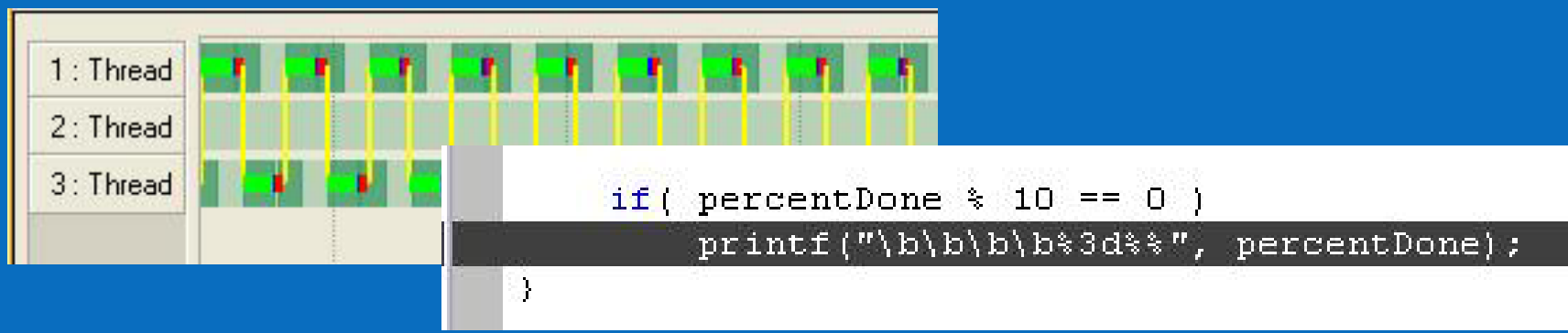


Software College

パフォーマンス

この実装には暗黙的な同期呼び出しが含まれる

コンテンション（競合）に起因するコンテキスト・スイッチによってパフォーマンスのスケーリングを制限する



設計段階に戻る

一般的なパフォーマンス問題

並列化によるオーバーヘッド

- スレッド作成によるスケジューリングなど

同期化

- 共有データの過度な使用、同じ同期化オブジェクトの競合

ロードバランス

- ワークロードの不適切な配分

粒度

- 不十分な実行単位

その他の問題 (VTune™ アナライザー)

- メモリー帯域、False Sharing

設計

目標

- コンテンション (競合) を除去する

プログレス更新を別々のスレッドで行う必要がある

タスクの並列処理をデータの並列処理に導入する

タスクの並列処理の設計

ステップ 1: 明示的なスレッディングを使用した素数生成のスレッド化

```
DWORD WINAPI FindPrimes(void *pData)
{
    PrimeData *p = (PrimeData *)pData;

    for( int i = p->start; i < p->end; i += p->skip )
    {
        if( TestForPrime(i) )
            p->primesFound++;

            gProgress++;
   }

    return 0;
}
```

タスクの並列処理の設計

ステップ 2: プログレス更新へのタスクの並列処理の追加

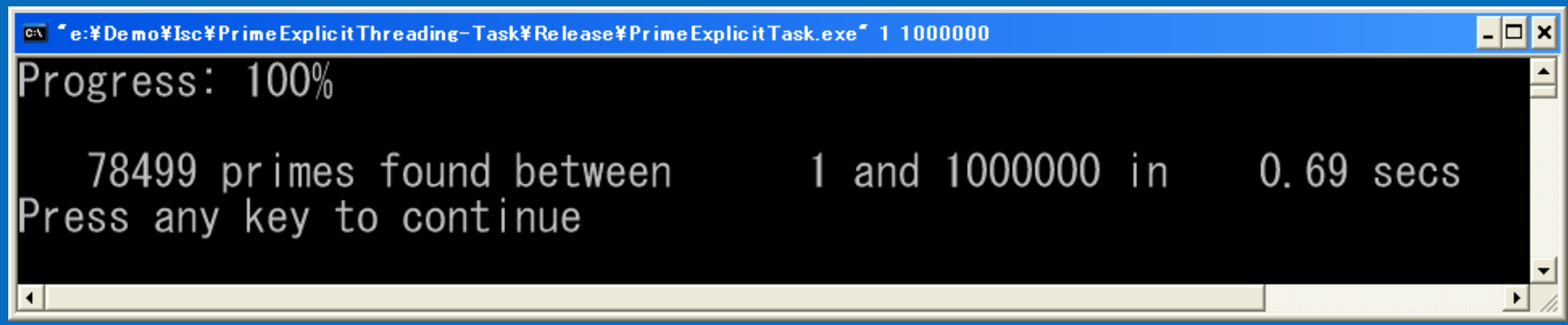
```
DWORD WINAPI ShowProgressThread(void *val)
{
int percentDone = 0, range = (int)val;

    while( percentDone < 100 ){
         percentDone = (int)((float)gProgress/(float)range *200.0f + 0.5f);

         printf("¥b¥b¥b¥b%3d%%", percentDone);
         Sleep(10);
    }

          return 0;
}
```

再設計された素数



スケーリングは 5.66/.69 = 8.2X

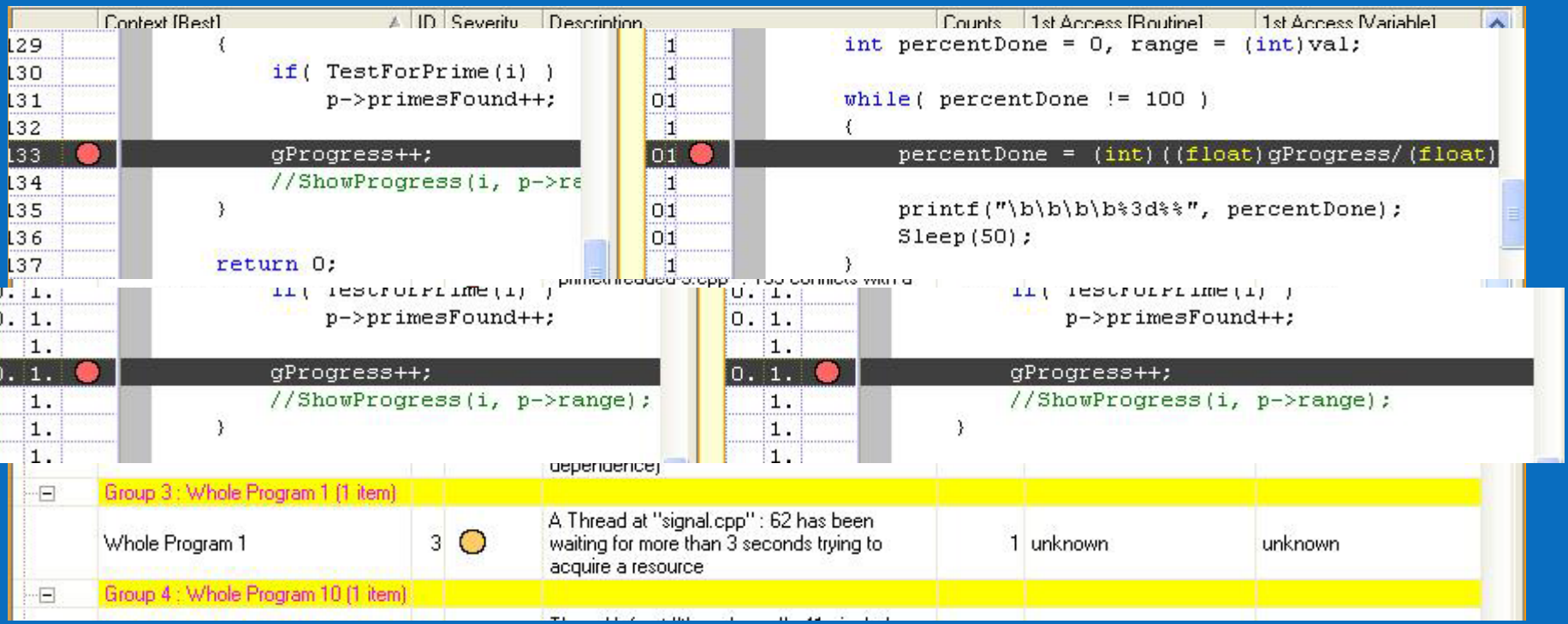
このアルゴリズムから期待できる最良のパフォーマンスか?

設計 – レビュー

必要な再設計の回数は?

別々のスレッドへの I/O 処理の割り当ては簡単にでき、パフォーマンスが向上する

正当性のデバッグの再確認

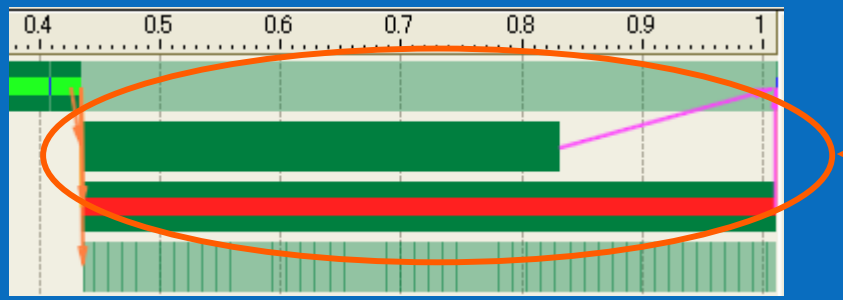


このバグは致命的か?

このバグは修正可能か?

その前にパフォーマンスをチェック!

パフォーマンス



ロード・インバランスが明白。2 つのスレッドで負荷が異なっている。

設計段階に戻る

設計の再確認

目標

- ロード･インバランスを解決する

データ分割のアルゴリズムを再確認して必要な変更を行う

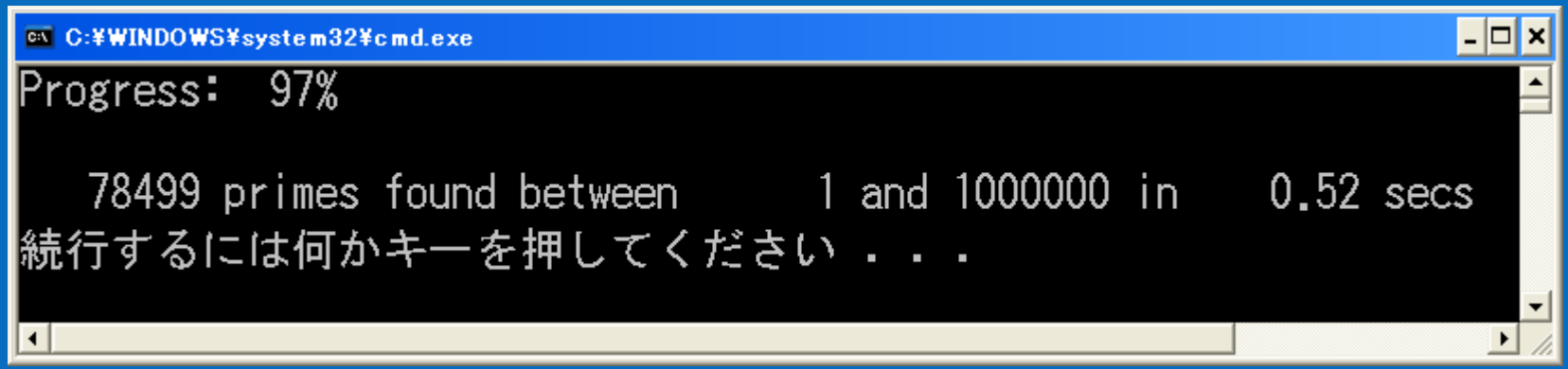
```
DWORD WINAPI FindPrimes(void *pData){
    PrimeData *p = (PrimeData *)pData;

   for( int i = p->start+2*p->thread; i < p->end; i += p->skip ){
        if( TestForPrime(i) )
            p->primesFound++;

            gProgress++;
    }

    return 0;
}
```

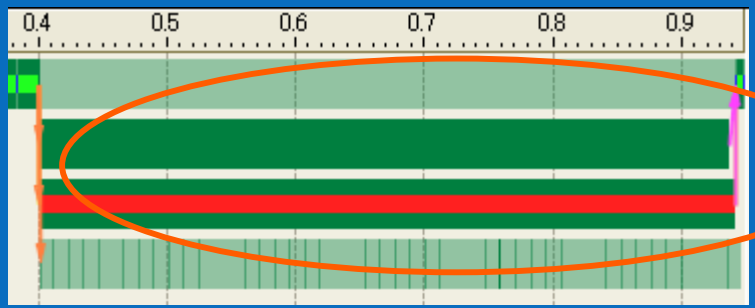
パフォーマンスの再確認



パフォーマンスのスケーリングは 5.66/0.52 = 10.8X

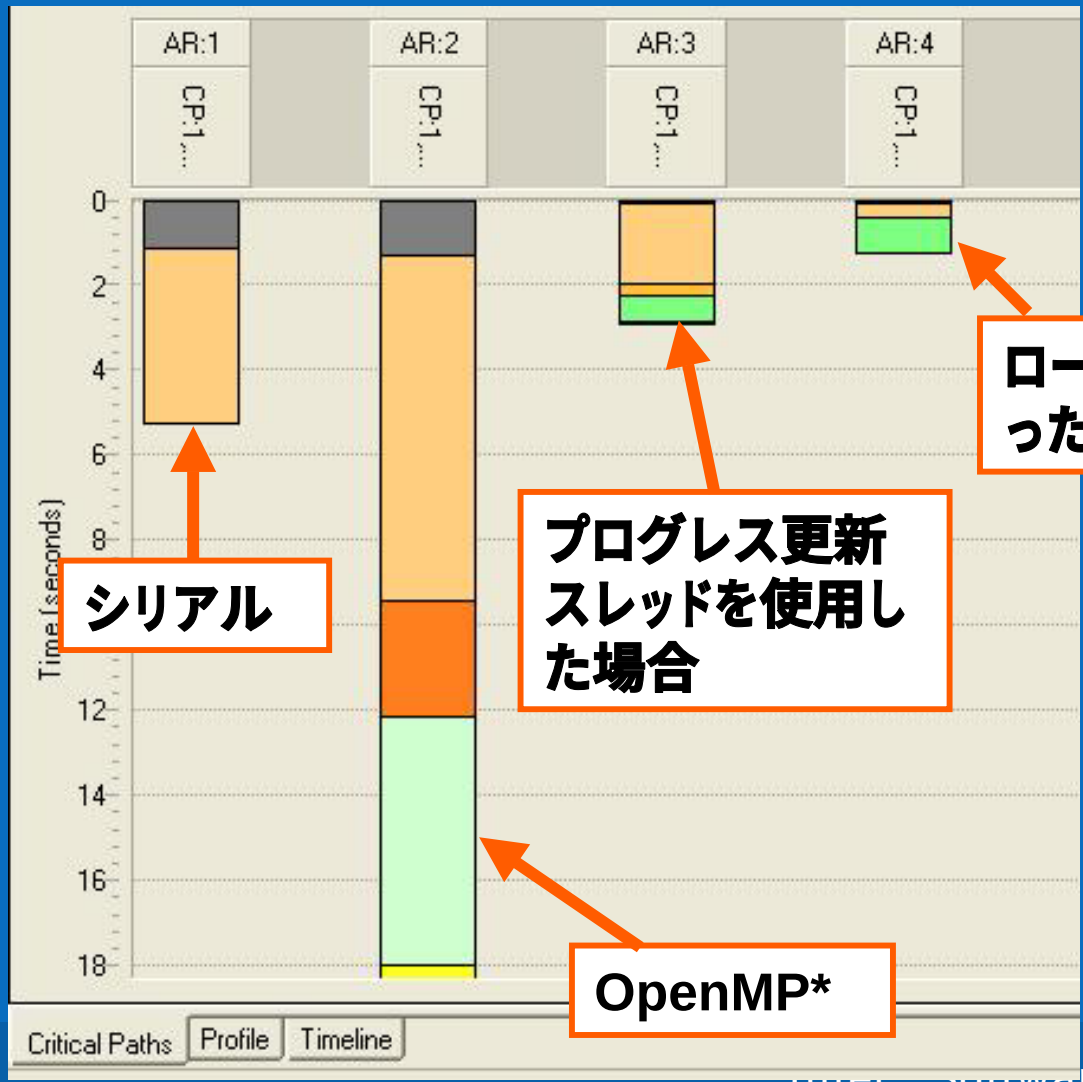
このアルゴリズムはプロセッサーが増加すると適切にスケールする

パフォーマンス

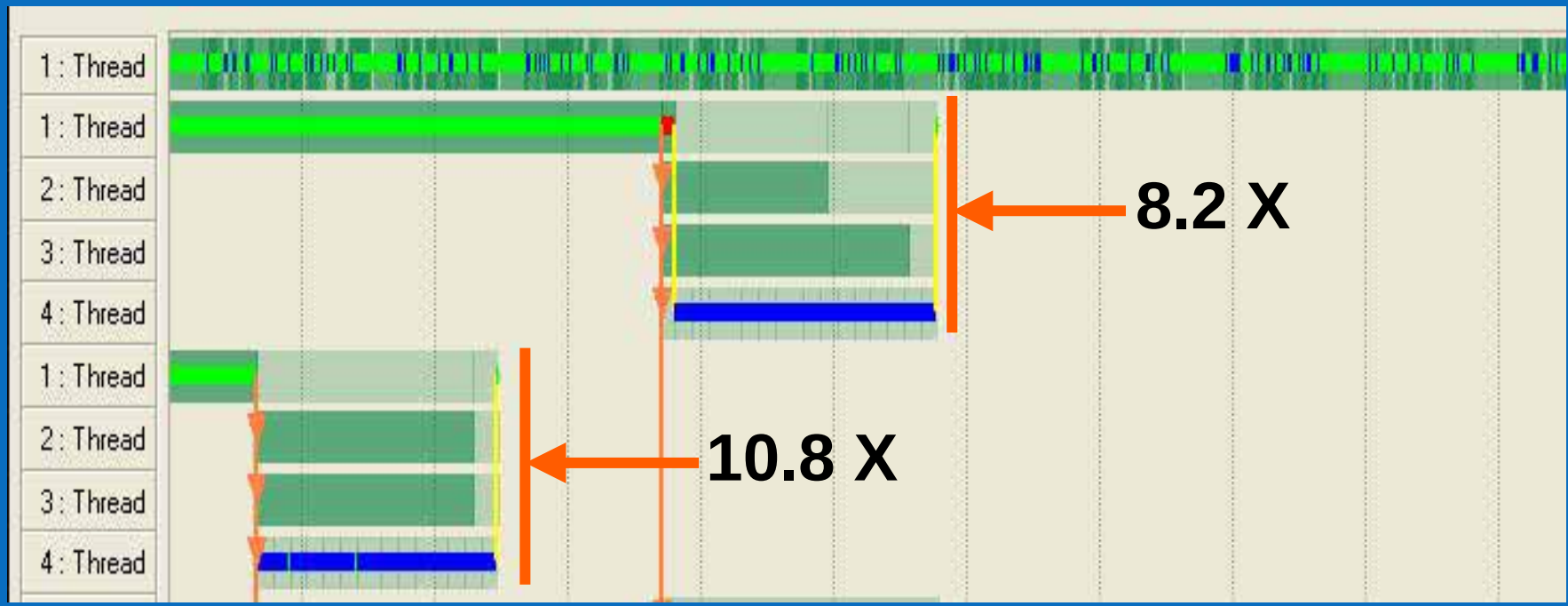


ロード・インバランス が解決されている。

比較解析



比較解析



アプリケーションをスレッド化するには、ソフトウェア開発サイクル全体を何度も繰り返す必要がある

質問の再確認

- どこをスレッド化すればいいのか?
- 選択した領域をスレッド化する価値はあるか?
- スレッド化に必要な時間は?
- 必要な再設計の回数は?
- どの程度のスピードアップを期待すべきか?
- パフォーマンスは期待値を満たすことができるか?
- プロセッサーを追加すれば性能が向上するか?
- どのスレッドモデルを使用すべきか?

まとめ

アプリケーションをスレッド化するには、設計、デバッグ、およびパフォーマンスのチューニングサイクルを何度も繰り返す必要がある

ツールを使用することで生産性を向上できる

デュアルコアおよびマルチコア・プロセッサーの能力を引き出す

並列化によるオーバーヘッド

スレッド作成のオーバーヘッド

- アクティブなスレッドの数の増加に従ってオーバーヘッドは急激に増加する

ソリューション

- 再使用可能なスレッドやスレッドプールを使用する
  - スレッド作成によるオーバーヘッドを吸収する
  - アクティブなスレッドの数を相対的に一定にする

同期化

ヒープ・コンテンション

- ヒープからの割り当ては暗黙的な同期化を引き起こす
- スタックに割り当てるか、スレッドローカル格納領域を使用する

アトミック更新とクリティカル・セクション

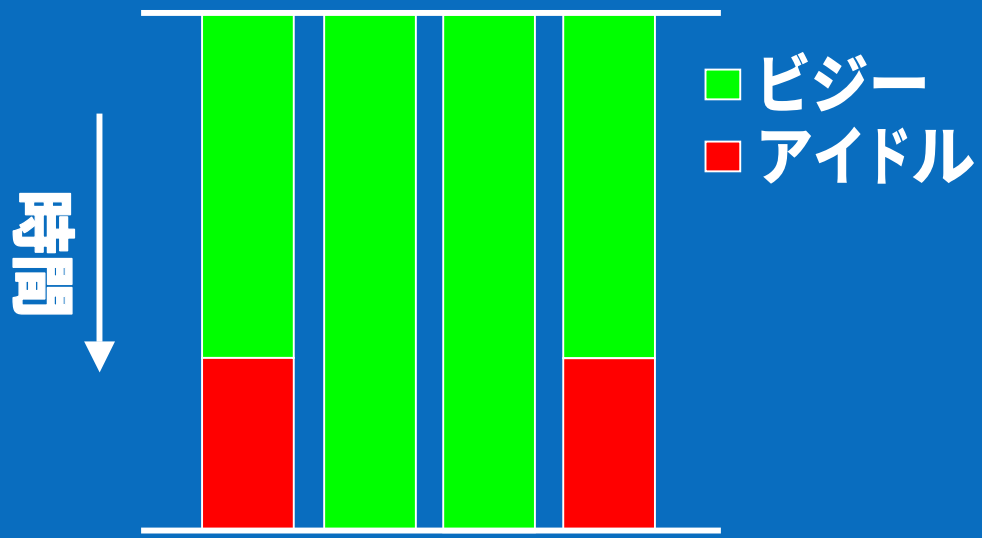
- いくつかの共有データ更新はアトミック演算 (インターロック・ファミリー) を使用可能
- 可能な場合は常にアトミック更新を使用

クリティカル・セクションと mutex

- クリティカル・セクション・オブジェクトはユーザー空間に存在する
- プロセス境界を超える可視性が必要ないときにクリティカル・セクション・オブジェクトを使用
- より少ないオーバーヘッド
- いくつかのアプリケーションで便利な可変スピンウェイト

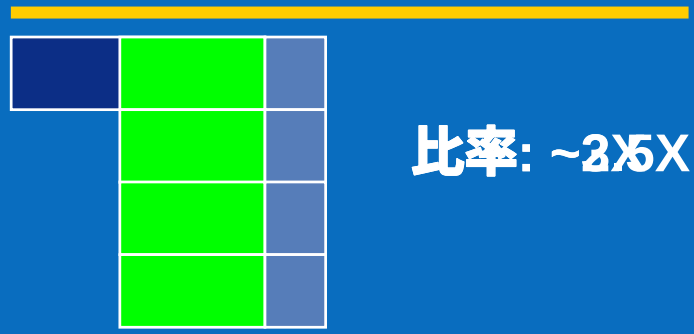
ロード・インバランス

不適切なワークロードはアイドルスレッドや
時間の浪費の原因となる



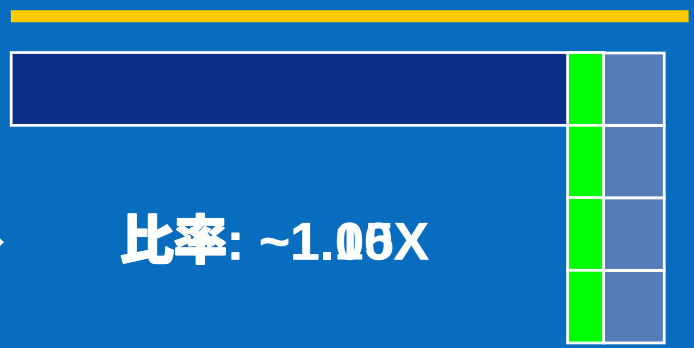
粒度

粗粒度



比率: ~3.5X

細粒度



比率: ~1.05X

本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。
製品に付属の売買契約書『Intel's Terms and conditions of Sales』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証 (特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的所有権を侵害していないことへの保証を含む) に関しても一切責任を負わないものとします。

インテル製品は、予告なく仕様が変更されることがあります。

* その他の社名、製品名などは、一般に各社の商標または登録商標です。