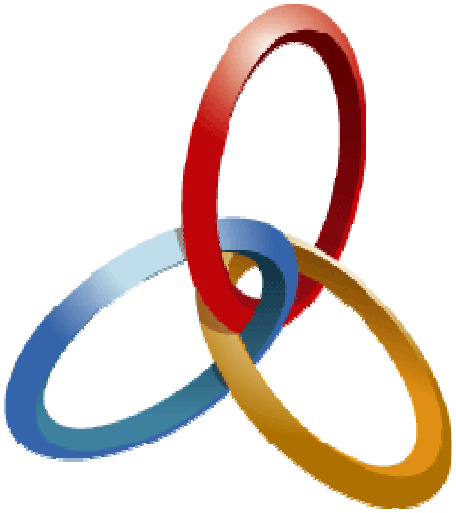


VSUG DAY 2008 Summer

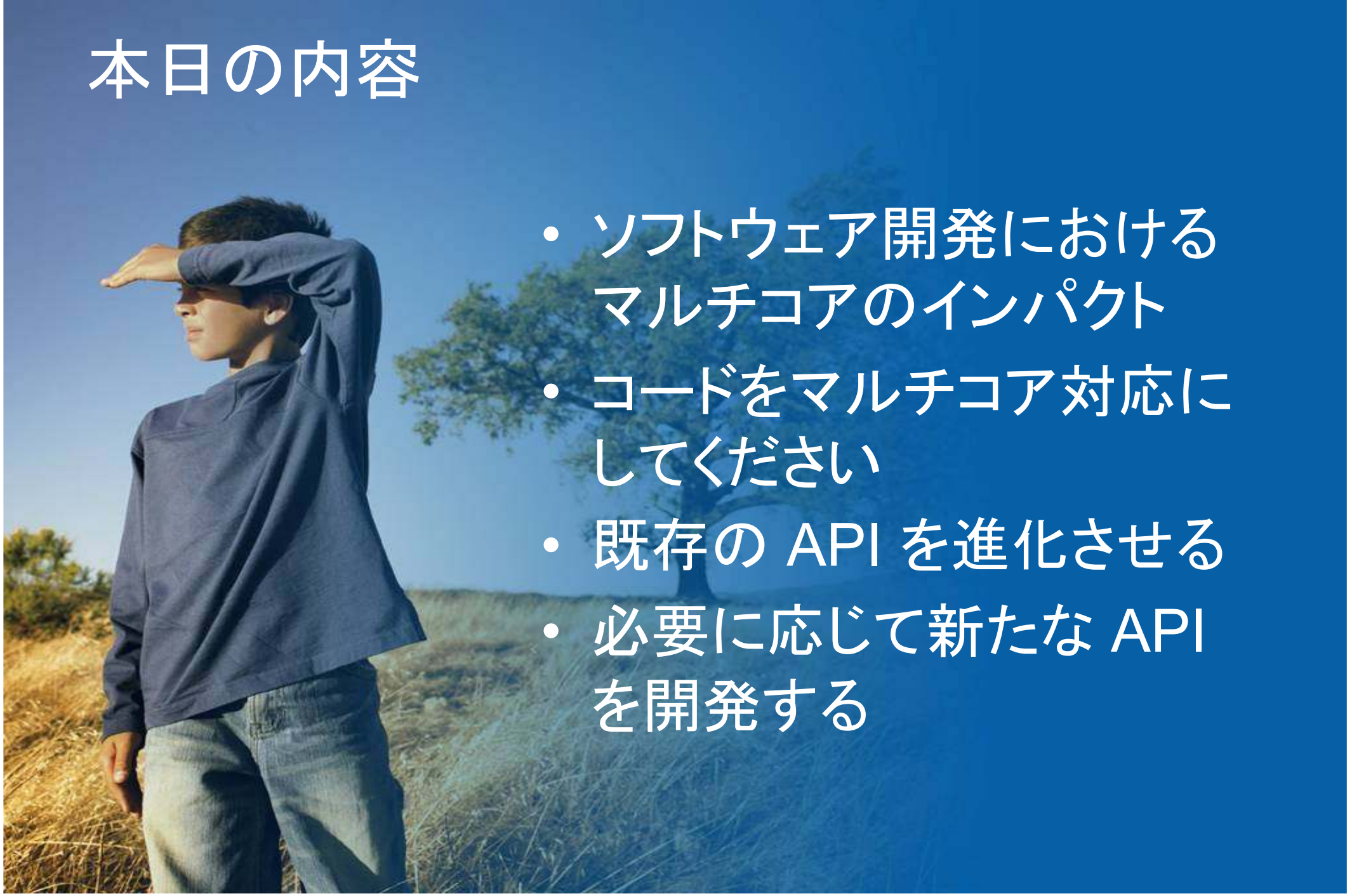
インテル® C++ コンパイラーによる マルチコア対応プログラム開発の未来

インテル株式会社
ソフトウェア＆ソリューションズ統括部
ソフトウェア技術部
菅原　清文

法務情報

- 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。本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスを許諾するためのものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』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) にも一切応じないものとします。インテル製品は、医療、救命、延命措置、重要な制御または安全システム、核施設などの目的に使用することを前提としたものではありません。
- インテル製品は、予告なく仕様や説明が変更される場合があります。
- 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、現在の予測に基づくものであり、予告なく変更される場合があります。
- 本資料で説明されているインテル製品には、不具合が含まれている可能性があり、公開されている仕様とは異なる動作をする場合があります。現在までに判明している不具合の情報については、インテルのサポートサイトをご覧ください。
- 性能に関するテストや評価は、一定のコンピューター・システム、コンポーネンツ、またはそれらを組み合わせて行ったものであり、このテストによるインテル製品の性能の概算の値を表しているものです。システム・ハードウェア、ソフトウェアの設計、構成等の違いにより、実際の性能は本サイトの性能テストや評価とは異なる場合があります。購入を予定してるシステムやコンポーネンツの性能については、他者の情報も併せて参照されることをおすすめします。インテル製品の性能評価についてさらに詳しい情報をお知りになりたい場合は、http://www.intel.com/jp/performance/resources/benchmark_limitations.htm を参照いただくか、1-800-628-8686 または 1-916-356-3104 (アメリカ合衆国) までご連絡ください。
- Intel、インテル、Intel ロゴ　は、アメリカ合衆国およびその他の国における Intel Corporation の商標です。
- * その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

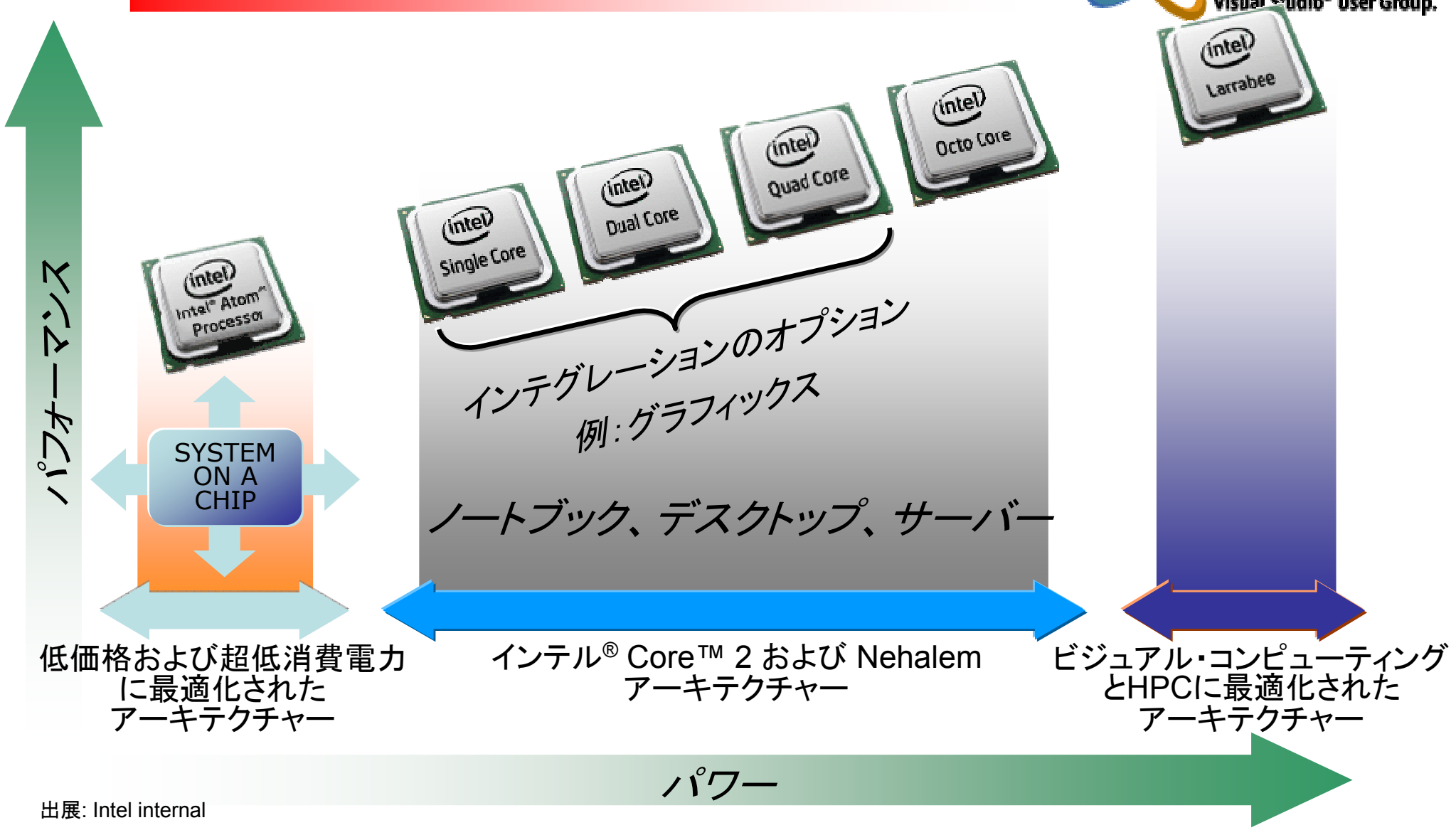
本日の内容



- ソフトウェア開発におけるマルチコアのインパクト
- コードをマルチコア対応にしてください
- 既存の API を進化させる
- 必要に応じて新たな API を開発する

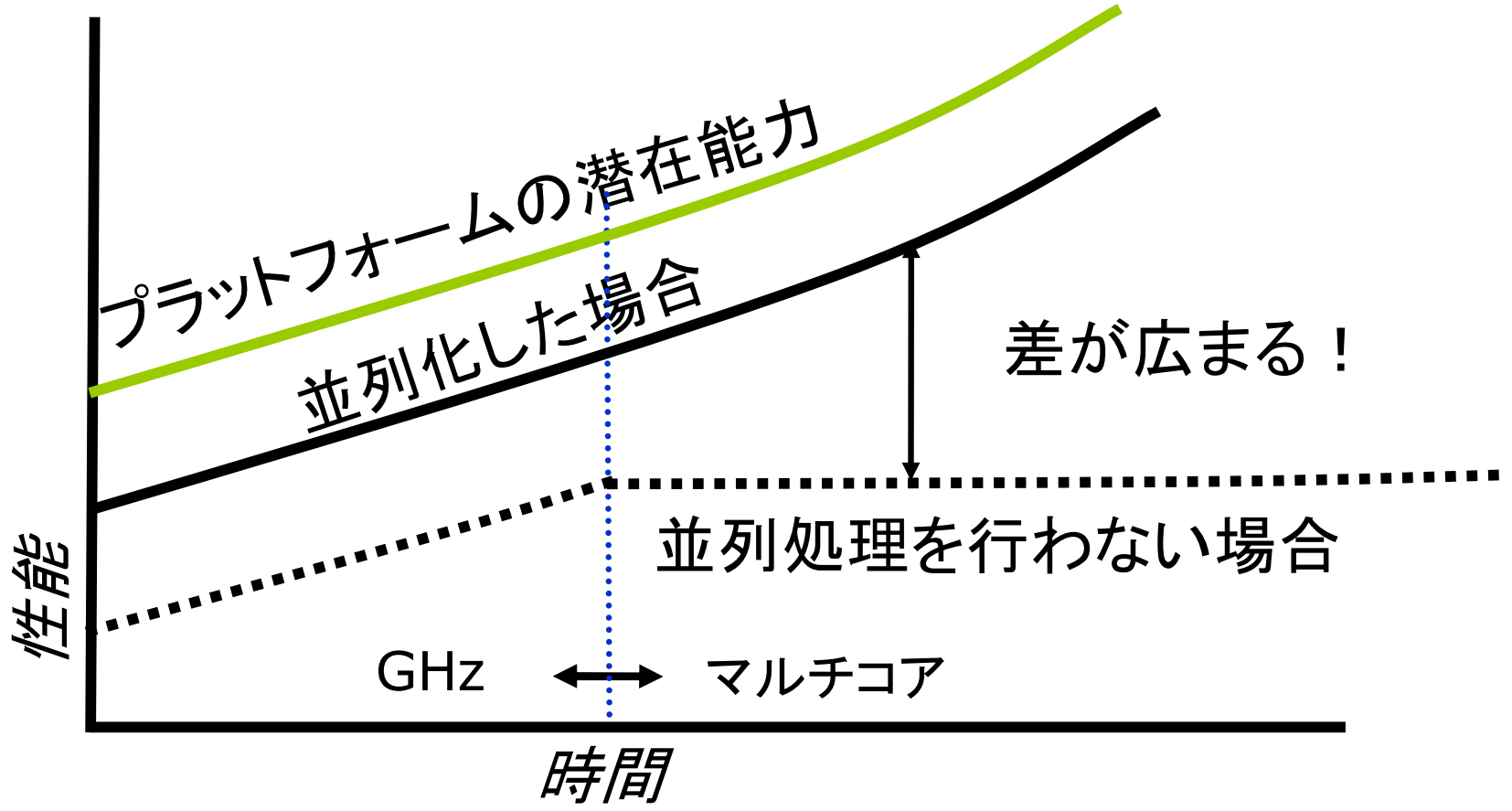
マルチコア・プロセッサー

ソフトウェア開発の概念を変えてください



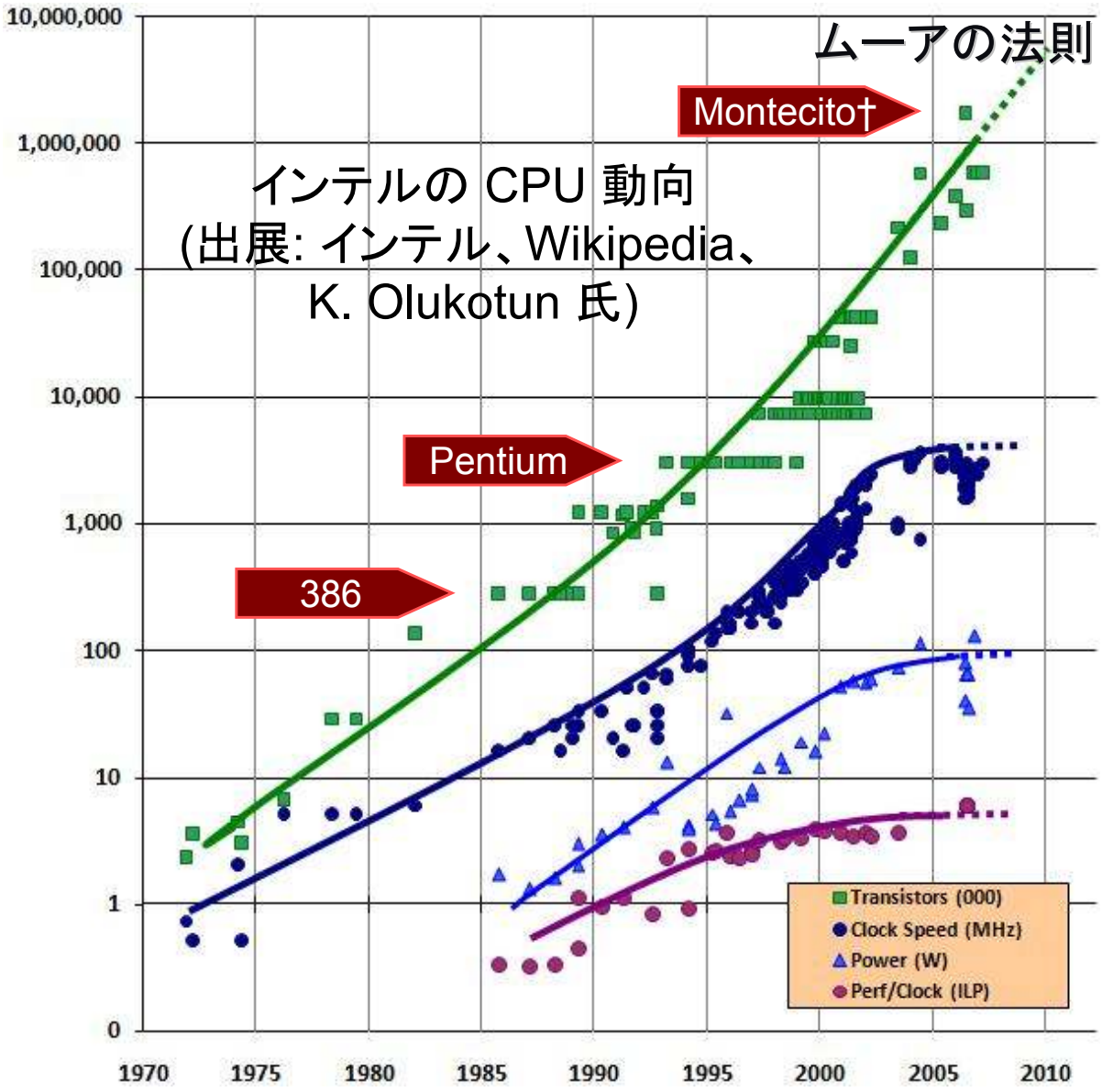
出展: Intel internal

並列処理の必要性



並列化は最良のパフォーマンスには必須

毎年~~速く~~多くなるプロセッサー



従来: より複雑なチップを利用してシングルストリームのパフォーマンスを向上

現在: チップあたりのコア数が増加 (GPU、NIC、SoC も提供)

フリーランチは終わった: 逐次アプリケーションとほとんどの並列アプリケーションでは変更が必要。並列処理を活用するアプリケーションが必要

世代の進化 > OO: “スレッド + ロック” プログラミング・モデルを超えるためには、世代の進化に重点をおく必要がある

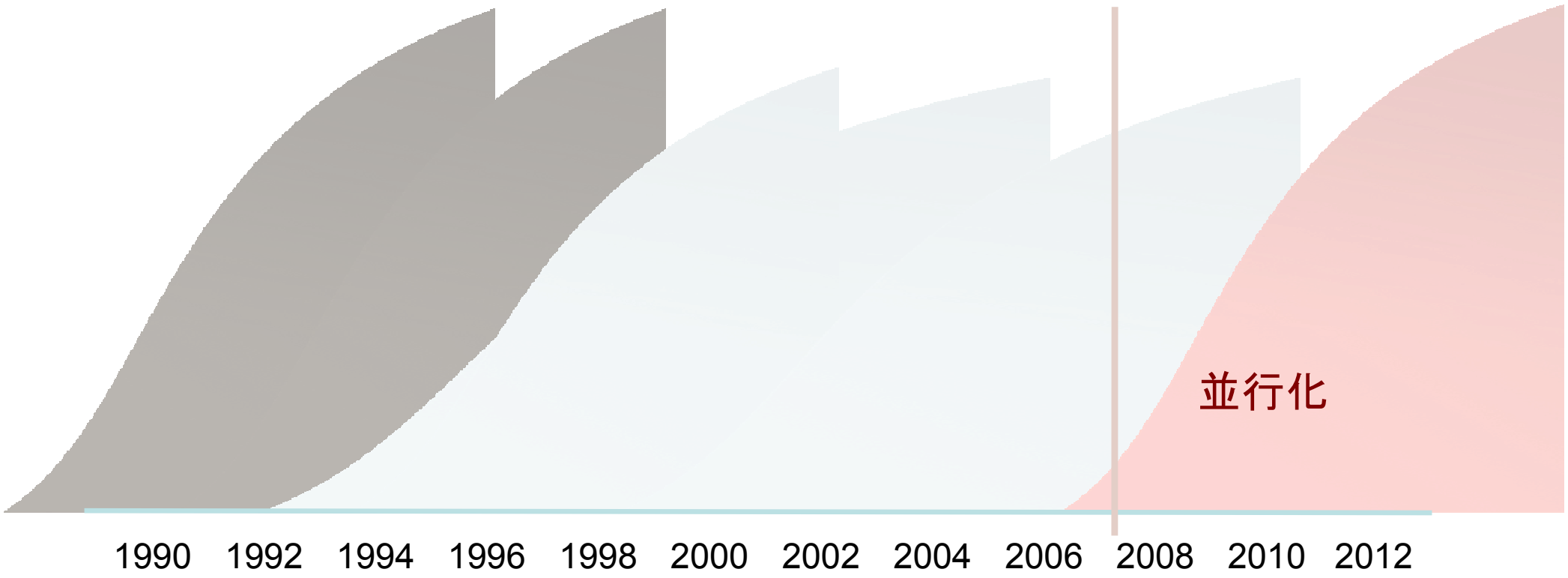
†開発コード名

問題（のほとんど）はクライアントにある

- “解決済み”: サーバー・アプリケーション（DB サーバー、Web サービスなど）
  - 多くの個別要求 – 要求ごとにスレッドを 1 つ割り当てることは容易
  - 同じコードの複数のコピーを実行することがほとんど
  - 通常は構造化データベースでデータを共有: トランザクションによる自動的かつ暗黙的な並行化制御
  - 一部に手を加えるだけで、“並列化問題が解決される”
- 未解決: 典型的なクライアント・アプリケーション（Photoshop などではない）
  - ユーザー "要求" ごとのスレッド数が多い
  - 同じコードの複数のコピーを実行することはほとんどない
  - 非構造化共有メモリーでデータを共有: エラーが発生しやすい明示的なロック – トランザクションは?

ソフトウェアにおける 6 つのトレンド

それぞれ 1958 年から 1973 年の間に誕生し、1990 年代/2000 年代に台頭し始めて、5 年強をかけて成熟したツール/言語/フレームワーク/ランタイム 体系が構築された



大きな相違点: 並行化時代の到来を察知している

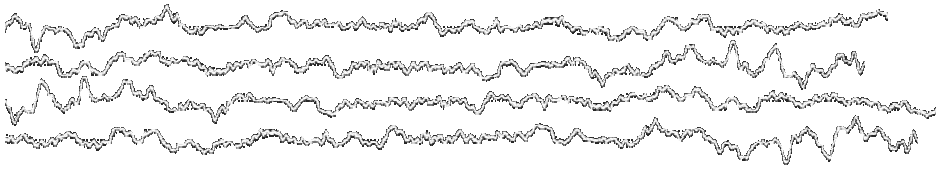
ソフトウェア全体に与える影響の比較

	GUI	オブジェクト	GC	ジェネリック	Web	並列化
アプリケーション・プログラミング・モデル	●●●	●●●	●	●	●	●●●
ライブラリーとフレームワーク	●●●	●●●		●●	●●●	●●●
言語とコンパイラー	●●	●●●	●	●●		●●
ランタイムと OS	●●		●●		●	●●●
ツール（設計、測定、テスト）	●●●	●	●		●●	●●

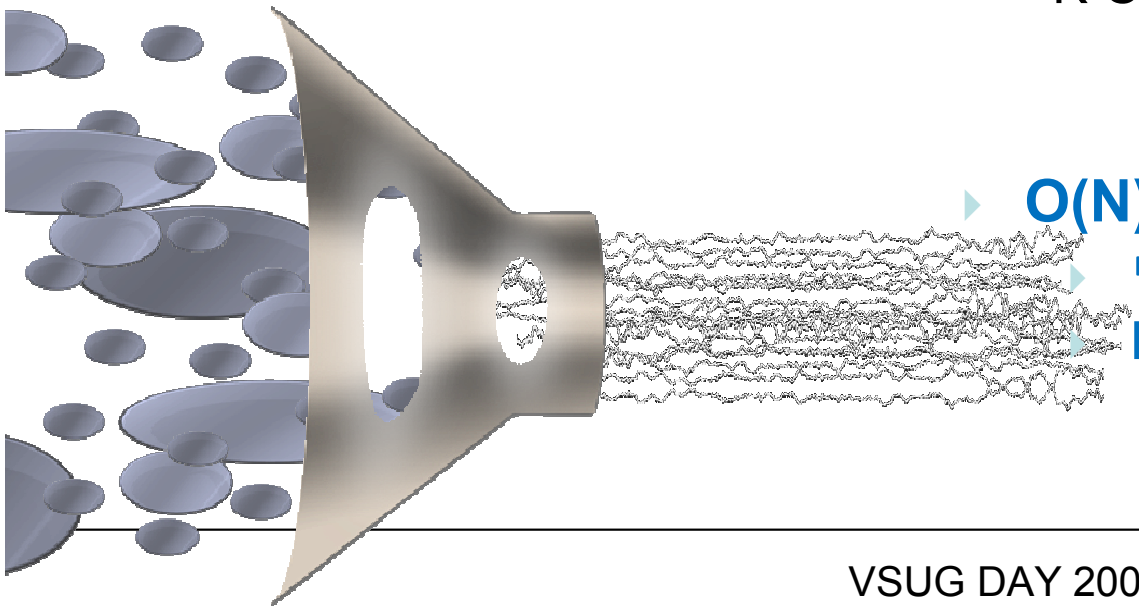
影響の範囲と重要性
● = 多少。1 つの主要な製品リリース
●● = 重要。1 つ以上の製品リリース
●●● = 新しい考え方/必須。複数の主要リリース

O(1)、O(K)、または O(N) の並列化?

- O(1): シーケンシャルと同様
  - フリーランチは終わり (CPU バウンドの場合): パフォーマンスは一定かわずかに向上
  - 潜在的に応答性は悪い



- O(K): 明示的に固定されたスループット
  - (指定される入力ワークロードに対して) K CPU 向けにハードウェア接続されたスレッド数
  - < K CPU の場合はパフォーマンスが向上せず、> K CPU の場合はスケーリングされない



- **O(N): スケーラブルなスループット**
  - **ワークロードはさまざまな作業に分割される**
  - **N コアまでマップ可能な並列化の活用**

内容

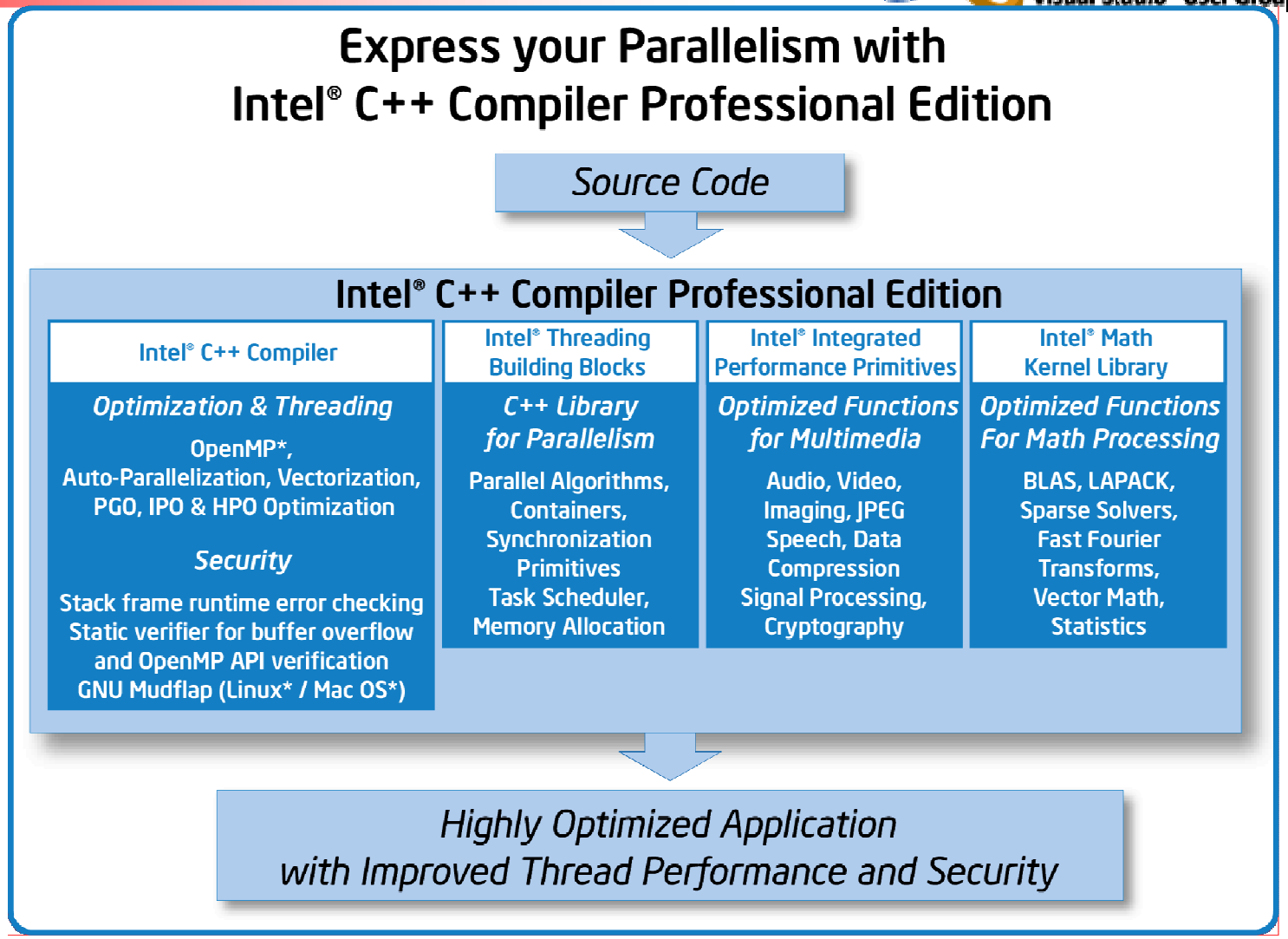
今日利用できるテクニック

- 今後考慮すべきこと
  - ソフトウェア・トランザクション・メモリー
  - ラムダ関数のサポート

- どのように並列化を実装するか？

インテル® C++ コンパイラー

- Windows 環境では Microsoft Visual C++* とソースおよびバイナリー互換
- Mac OS* X および Linux 環境では gcc とソースおよびバイナリー互換



今日利用可能なテクニック

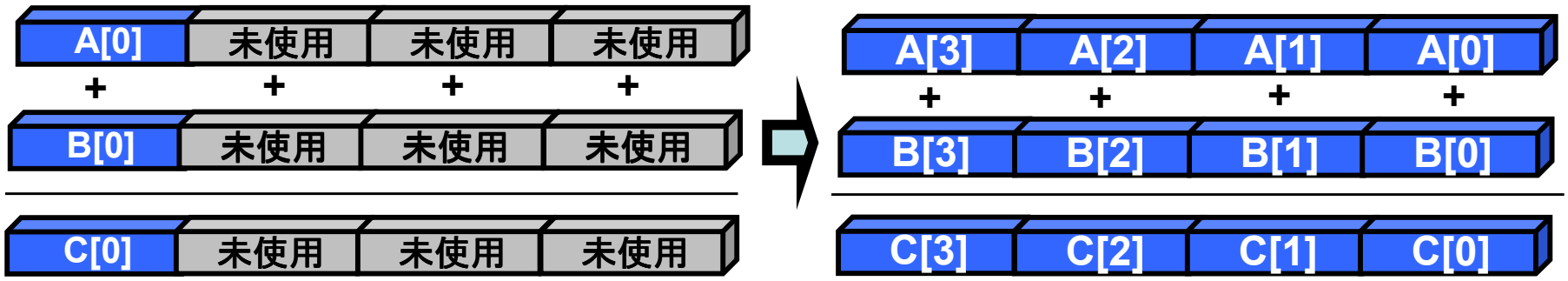
1. 自動ベクトル化
   - SIMD 命令の利点を活用
   - データ並列、ベクトル化のヒント
   - イントリンジック（組み込み関数）
2. ループレベルの並列化
   - 自動並列化（/Qparallel）
   - 並列化のヒント
3. OpenMP
4. スレッディング・ビルディング・ブロック
5. ネイティブ・スレッドとデバッガーのサポート
   - 詳細な制御が可能であるがより作業が要求される
   - 並列化をサポートする既存のライブラリーを活用

1. 自動ベクトル化

- 命令レベルの並列性
  - SSE 命令を生成する
  - 同時に複数の要素を操作
    （2 double、4 float そして 4 int など）

例:

```
for (int i=0;i<N;i++)
        c[i] = a[i]+b[i];
```

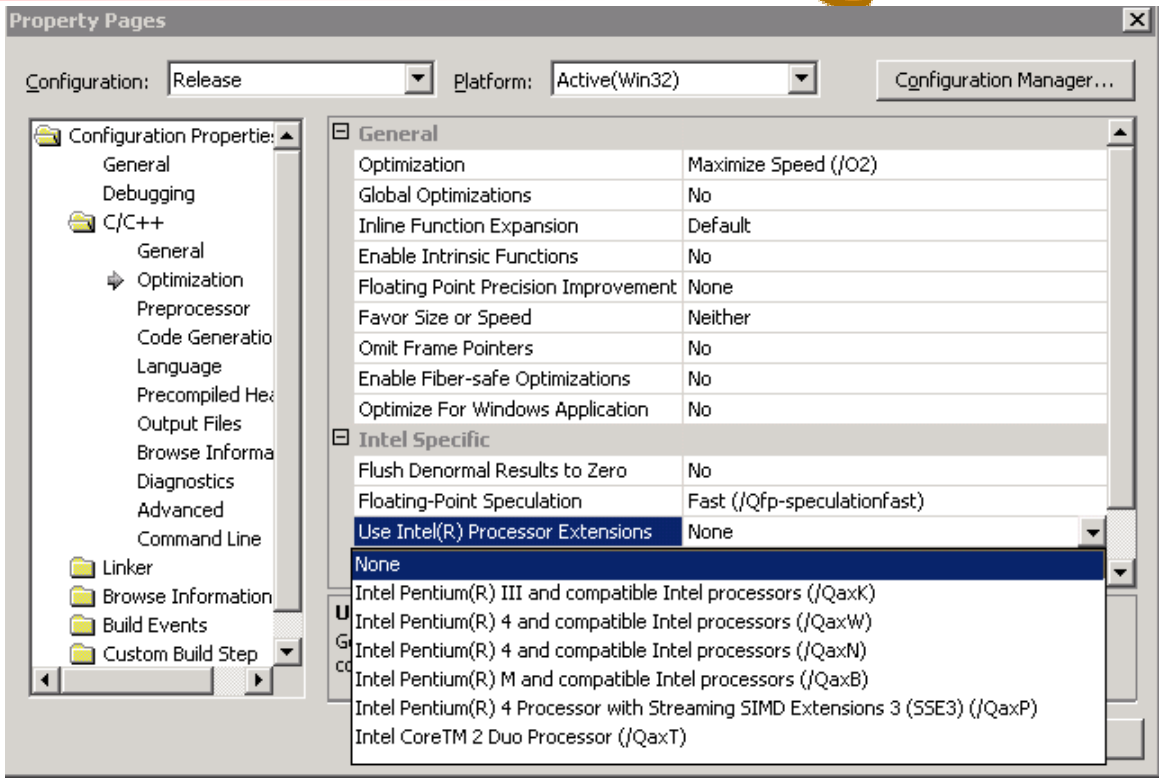


スカラー操作　　ベクトル操作

A, B と C は 128 ビット SIMD レジスター

自動ベクトル化（続き）

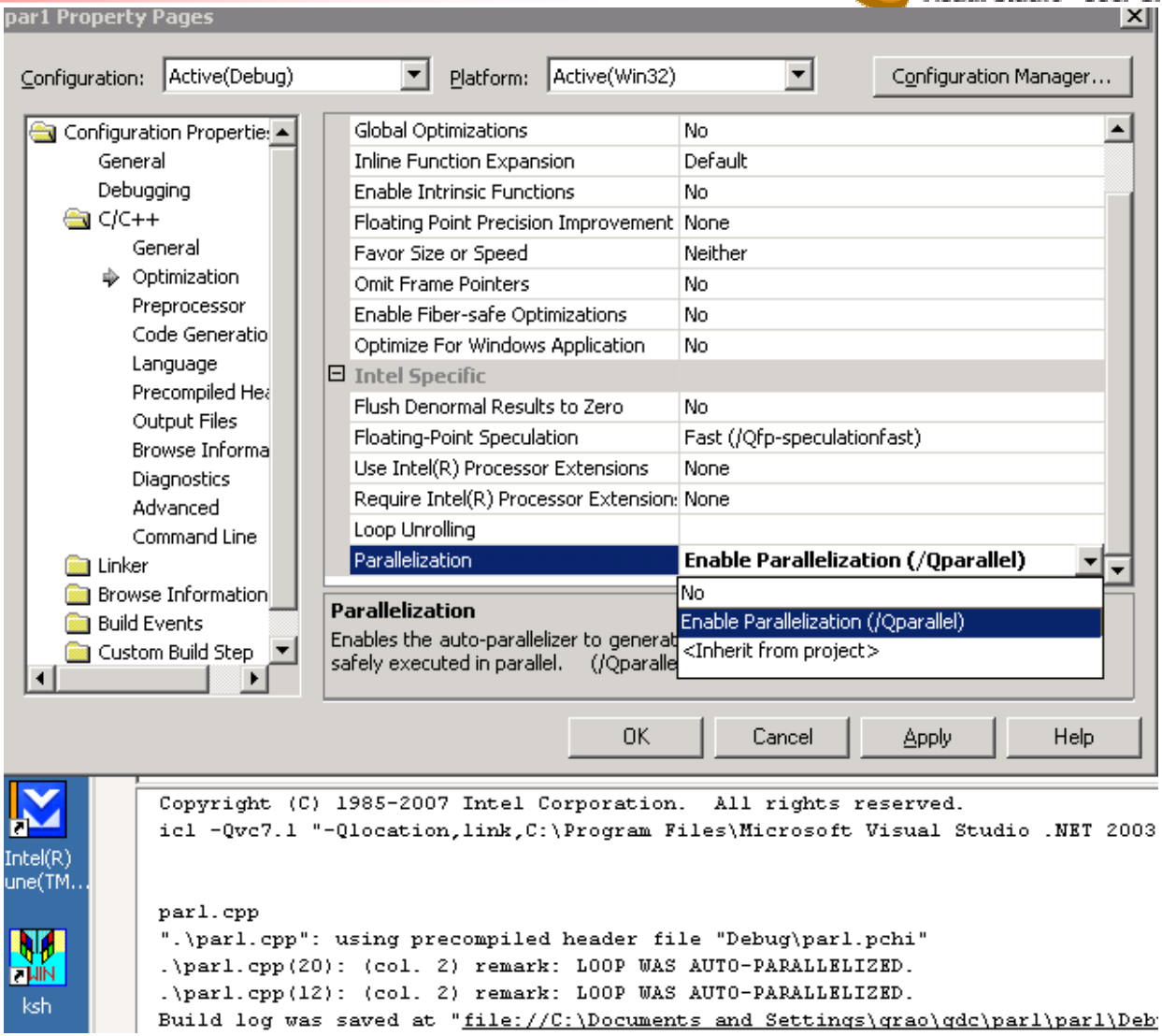
- /Qx* もしくは /Qax* オプションで有効となる
- ビルドログ中の “Loop vectorized” メッセージを参照



- ベクトル化できない場合
  - コンパイラーがループを自動ベクトル化するには、ループは独立していなければいけない
  - /Qvec-report[n] オプションを使用してループ最適化の詳細レポートを表示する（n は 1、2、3、4、5）

2. 自動並列化 : /Qparallel

- /Qparallel オプション
- /Qpar-report[n] で並列化のレポートを表示（n は1、2、3）



自動並列化: どのように作用するか

- ループレベルの並列化

```
#define N 10000
float a[N], theta[N], r[N];
void f1()
{
        a[0] = 0;
        for(int i = 1; i < N; i++) {
                a[i] = r[i]*cos(theta[i]);
        }
}
```

```
.¥par1.cpp(20): (col. 2) remark: LOOP WAS AUTO-PARALLELIZED.
.¥par1.cpp(13): (col. 2) remark: LOOP WAS AUTO-PARALLELIZED.
```

- ループ内でデータの依存性がある場合、ループを再構成する

結果は保証されるが考察の余地はある

スレッドとベクトル・レベルの並列化


```
float a[M][N], b[M][N+1], c[M][N], d[M][N];
for (j = 0; j<N; j++)
  for (i = 1; i<M; i++)
    a[i][j] = a[i][j] + x;
    b[i][j+1] = a[i][j] + b[i][j]; // J のループには配列 b のフロー依存がある
    d[i][j] = b[i][j+1] + c[i][j] + d[i][j];
  }
}
```

ループ変換、ループ分割後の
ループ並列/ベクトル化

```
parallel for (i=0; i<M; i++) {
  a[i][1:N] = a[i][1:N] + x;
  for (j=0; j<N; j++) {
     b[i][j+1] = a[i][j] + b[i][j];
  }
}
```

```
parallel for (i=0; i<M; i++) {
  d[i][1:N] = b[i][2:N+1) +
        c[i][1:N) + d[i][1:N];
}
```

ループ変換、並列化、ベクトル化におけるより良い相互関係

3. OpenMP*

- ソース中に #pragma 宣言子を挿入し並列化を明示
- 例:

```
#include <omp.h>
…
#pragma omp parallel for
for (;;;) {//stuff}
```

- /Qopenmp オプションを指定しコンパイル
- インテル® コンパイラー 10.1 は Microsoft* VS 2005 とソースおよびバイナリー・レベルで互換性保持
  - 以前のバージョンでは OpenMP バイナリーの一部互換性がなかった
- システム・レベルの性能を向上させるため、スタティック・ライブラリーではなく dll を利用する
- OpenMP の利用法は簡単
  - スレッドの生成、スケジュール、同期
  - ループ空間の分割
  - データ共有
- 現在のコンパイラーは OpenMP 2.5 仕様をサポート
- http://www.openmp.org

既存のアプリケーションをスレッド化する
グローバル変数アクセス診断

- 問題： 多くのグローバル変数を持つアプリケーションをスレッド化する場合、スレッド間での変数アクセスを保護する必要がある

```
> type a.cpp

 1:     static int x;
 2:     void foo(int *);
 3:     void funcx(void){
 4:       int y;
 5:       x=2;
 6:       y=x;
 7:       foo(&x);
 8:     }
 9:
10:     extern int q;
11:     int p;
12:     void funcy(void) {
13:        q=10;
14:         p=5;
15:     }
```

```
> icl /Qww1710,1711,1712 a.cpp
a.cpp(5): warning #1711: assignment to
   statically allocated variable "x"
  x=2;
a.cpp(6): warning #1710: reference to
   statically allocated variable "x"
 y=x;
a.cpp(7): warning #1712: address taken
   of statically allocated variable "x"
 foo(&x);
 a.cpp(13): warning #1711: assignment
   to statically allocated variable "q"
 q=10;
a.cpp(14): warning #1711: assignment to
   statically allocated variable "p"
 p=5;
```

OpenMP の利用例


```
void sp_1a(float a[], float b[], int n)
{
  int i;
  #pragma omp parallel shared(a,b,n) private(i)
  {
  #pragma omp for
      for (i = 0; i < n; i++)
        a[i] = 1.0 / a[i];
  #pragma omp single
        a[0] = MIN( a[0], 1.0 );
  #pragma omp for nowait
      for (i = 0; i < n; i++)
        b[i] = b[i] / a[i];
   }
}
```

既存の API を進化させる

OpenMP 2.5 は単純なポインター追尾のループを並列化できない

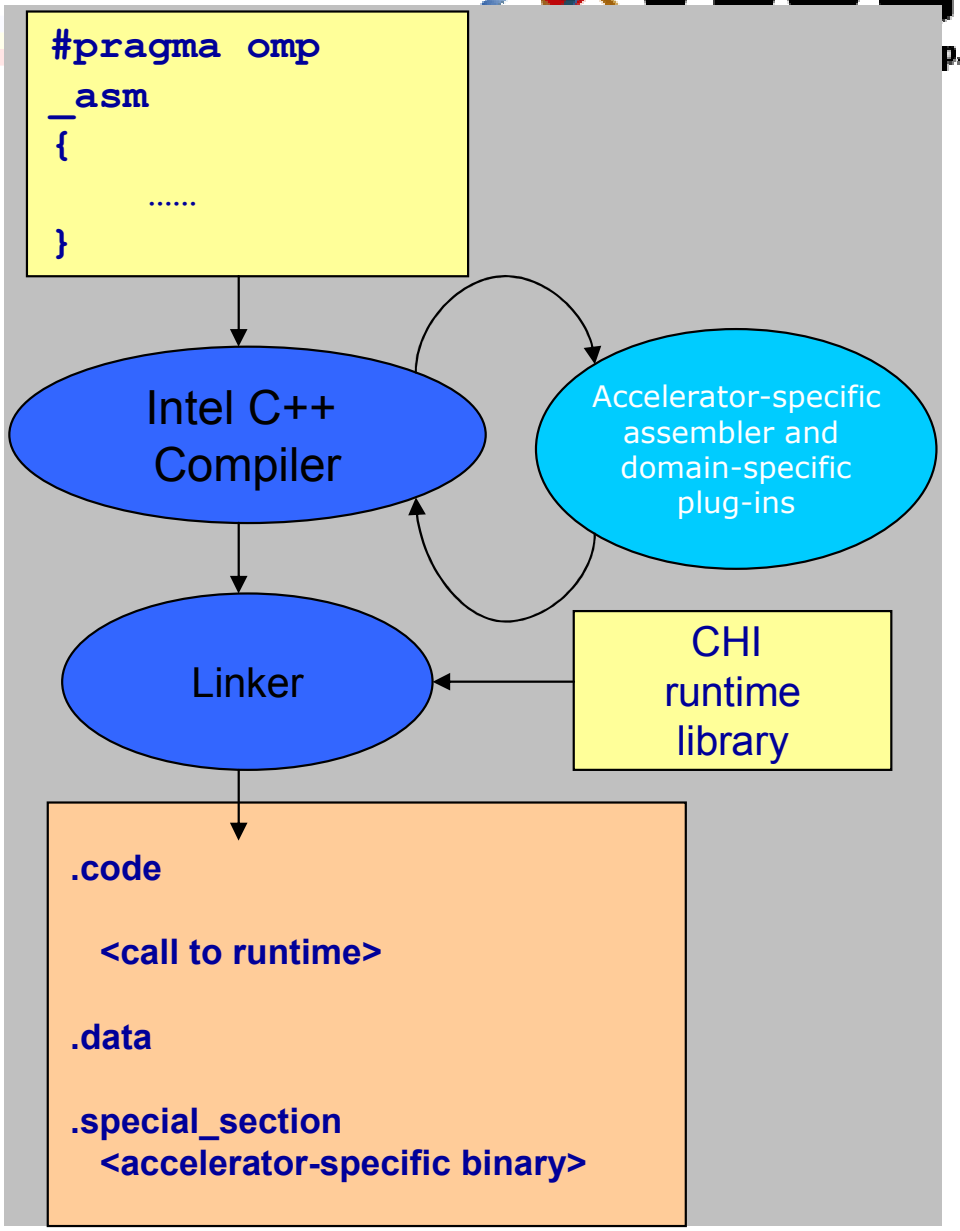
```
nodeptr list, p;

For (p=list; p!=NULL; p=p->next)
      process(p->data);
```

OpenMP 3.0 では、タスク句によって上記の問題を解決

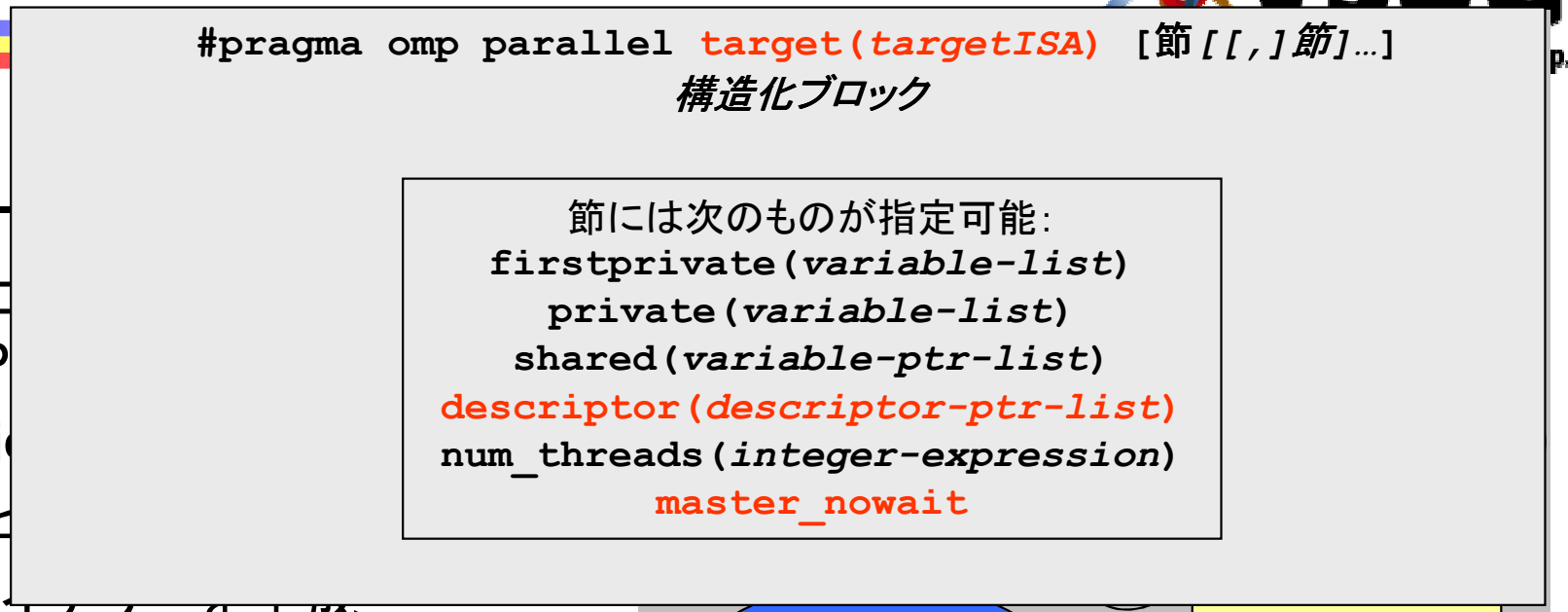
```
nodeptr list, p;

#pragma omp parallel
{
     #pragma omp single
     {
          for (p=list; p!=NULL; p=p->next)
          #pragma omp task firstprivate(p)
           process(p->data);
     }
}
```

EXOCHI プログラミング環境

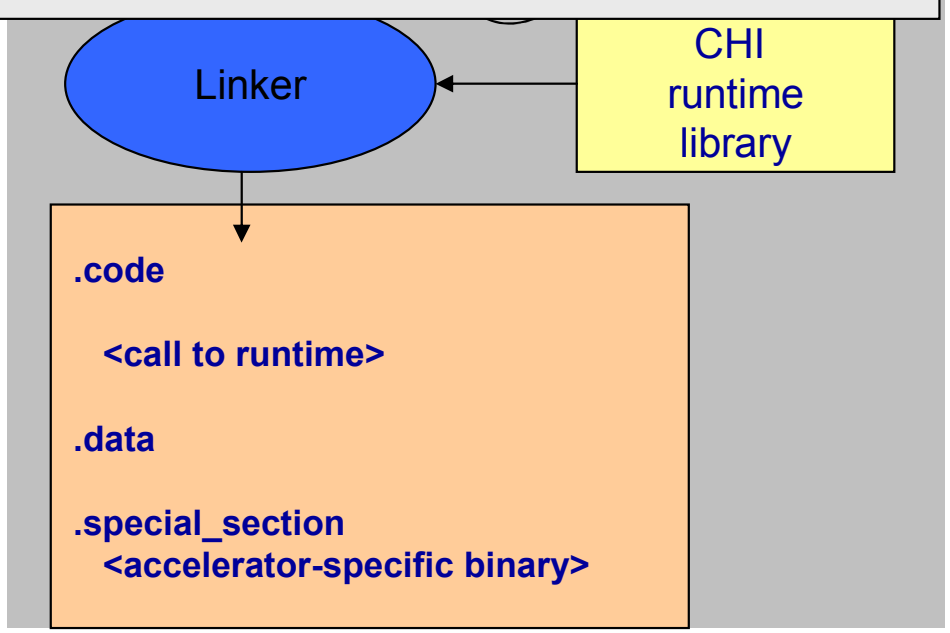
- コンパイラー
  - フロントエンドと OpenMPプラグマを変更
    - Fork/join
    - 生産と消費並列化
  - 大きなバイナリーを生成
- CHI ランタイム
  - マルチ・シェレディング：ユーザー・レベルのスレッド化
  - 異種コアのマルチ型への拡張
    - 例：Intel GMA X3000
    - 例：通信処理のためのデータストリーミング・シストリック・アレイのアクセラレーター



EXOCHI プログラミング環境



- コンパイラー
  - フロントエ
    OpenMP
    - Fork/j
    - 生産と
  - 大きなバイナリーを生成
- CHI ランタイム
  - マルチ・シェレディング : ユーザー・レベルのスレッド化
  - 異種コアのマルチ型への拡張
    - 例 : Intel GMA X3000
    - 例 : 通信処理のためのデータストリーミング・シストリック・アレイのアクセラレーター



4. インテル® スレッディング・ビルディング・ブロック(TBB)

- ヘッダーとランタイム・ライブラリーからなる
- タスクベースの並列化を表現する
  - スケーリングを持ち、将来における保守作業を軽減
  - 複数のスレッディング向けの置き換えライブラリー
- オープンソース版のヘッダーを入手可能
- http://www.threadingbuildingblocks.org/

インテル® TBB アーキテクチャー

- C++ の STL に似たアーキテクチャー
- .h ファイル中に並列アルゴリズムが定義される
  - アルゴリズムのテンプレート:
    parallel_for, parallel_reduce, parallel_scan
  - アルゴリズムに必要な引数:
    1. ループ空間の範囲
    2. 実行されるタスクの関数オブジェクト
  - C プログラマー向けか？
    - 関数オブジェクト = functor = C++ で関数へのポインターを書き込む方法
    - コンパイラーは関数へのポインターと異なる、functor の最適化を得意とする

インテル® TBB 例題

```
using namespace tbb;
using namespace std;
int main() {
   task_scheduler_init init;
   …
   parallel_for(blocked_range<size_t>(0,to_scan.size(),100),
   MyTask(to_scan, max, pos));
   …
}
class MyTask {
public:
   void operator() ( const blocked_range<size_t>& r) const {
       for ( size_t i = r.begin(); i != r.end; i++) { … }     }
   MyTask(string &s, size_t *m, size_t *p) {…} //元のコード
};
```

5. ネイティブ・スレッド

- Createthread() 関数を呼び出しスレッドを生成
- 同期とスレッド待機は開発者によって明示的に管理される
- ソースとバイナリーは VS2008 と互換
- 生成されたコードは Visual Studio* .NET デバッガーと互換性あり
  - OpenMP*、TBB、自動ベクトル化および自動並列化を利用したコードに対しても有効
  - その他の選択肢： インテル・デバッガー
    - インテル・コンパイラーに同梱される

並列化プログラミング

内容

今日利用できるテクニック

- –今後考慮すべきこと
  - _parallel, _spawn
  - ソフトウェア・トランザクション・メモリー
  - ラムダ関数のサポート
- –どのように並列化を実装するか？

__parallel, __spawn

- 新たな言語拡張
- Fork-join モデルによる構造化非同期実行
- OpenMP ライブラリーを流用したより単純なサブセットを提供
  - *__parallel*
    - 指示されたステートメント内でタスクの並列実行環境をセットアップ
    - 並列ステートメントの終了後シリアル実行が再開される
  - *__spawn*
    - __parallel スコープ内で起動される
    - ステートメント内のタスクを並列に実行する
    - 複数の __spawn ステートメントで並列性を増やす
  - *__par* で並列に実行するループを明示
  - *__critical* でクリティカル・セクションを明示
  - /Qpar オプションで言語拡張を有効にする

インテル® C++ 並列探索コンパイラー、プロトタイプ版

Whatif.intel.com で入手可能

__parallel を利用する

- OpenMP をよりシンプルに
- 要件はより簡単
  - アプリケーションはデータ共有やプライベート変数を要求しない、もしくはわずかしか使用しない
- 開発者は正当性を保証しなければいけない！

__parallel == #pragma omp parallel
__spawn == #pragma omp task
__critical == #pragma omp critical
__par for == #pragma omp parallel for

インテル® C++ 並列探索コンパイラー、プロトタイプ版
Whatif.intel.com で入手可能

例題1：__parallel、__spawn


```
__parallel {
  __spawn f_sum(500, a, b, c);
  __spawn f_sum(500, a+500, b+500, c+500);
}
int a[1000], b[1000], c[1000];
void f_sum ( int length, int *a, int *b, int *c )
{
  for (int i=0; i<length; i++)
       c[i] = a[i] + b[i];
}
```

インテル® C++ 並列探索コンパイラー、プロトタイプ版

Whatif.intel.com で入手可能

ソフトウェア・トランザクション・メモリー（STM）

- Lock ベースのプログラミングに代わる手法
- lock や mutex を利用するモデルでは；
  - データをアクセスする前にロックを取得
  - 多くの場合同期は必要ない
  - 無駄なロックと解放は、パフォーマンスに影響！
- STM を利用する際の原則
  - 多くのロックは競合しない
  - 多くの場合アクセスは読み込みのみ
  - メモリーアクセスはトランザクション
  - トランザクションは競合で退却

インテル® C++ STM コンパイラー、プロトタイプ版
Whatif.intel.com で入手可能

STM の利用法

- __tm_atomic
  - 言語拡張はトランザクションをスコープする
- __declspec(tm_callable)
  - 関数の修飾
  - 関数は __tm_atomic スコープから呼び出し可能
  - C++ クラスのメンバー関数を含む
  - 関数は競合で退却
- __declspec(tm_waveable)
  - 競合が起こっても関数を退却しない
- バイナリー互換
  - 非 STM ソースとの混在や結合可能

インテル® C++ STM コンパイラー、プロトタイプ版
Whatif.intel.com で入手可能

STM 例題

```
__tm_atomic {
      x = new BaseXXX();
      x->TxnAddOne();
} //並列領域
```

```
class BaseXXX {
   public:
__declspec(tm_callable)
   virtual void TxnAddOne()
   {
     xx = xx + 1;
   }
};
```

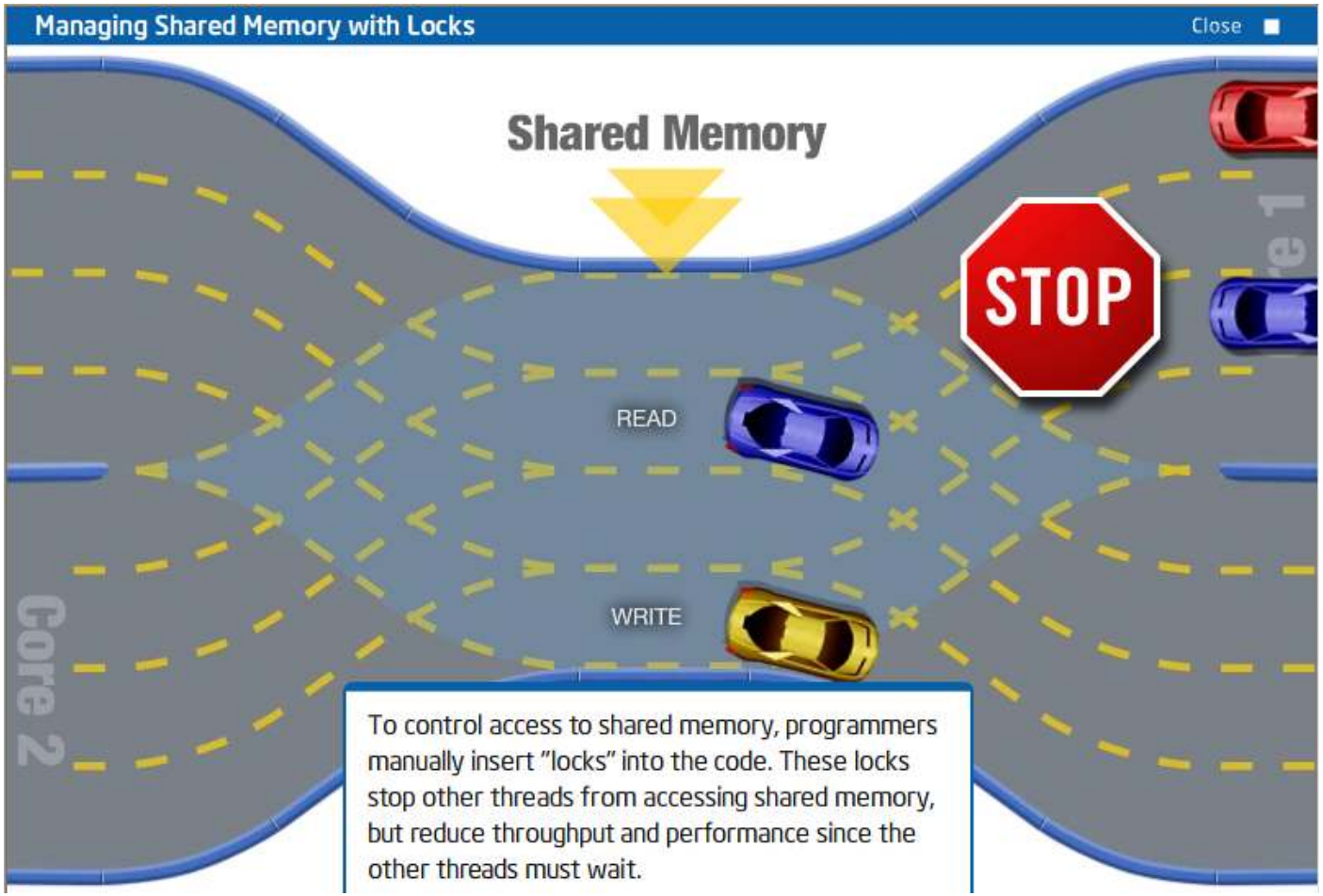
- /Qtm_enabled
  オプションで有効

インテル® **C++ STM** コンパイラー、プロトタイプ版
Whatif.intel.com で入手可能

インテル® C++ STM コンパイラーのコンセプト

VSUG
Visual Studio® User Group.

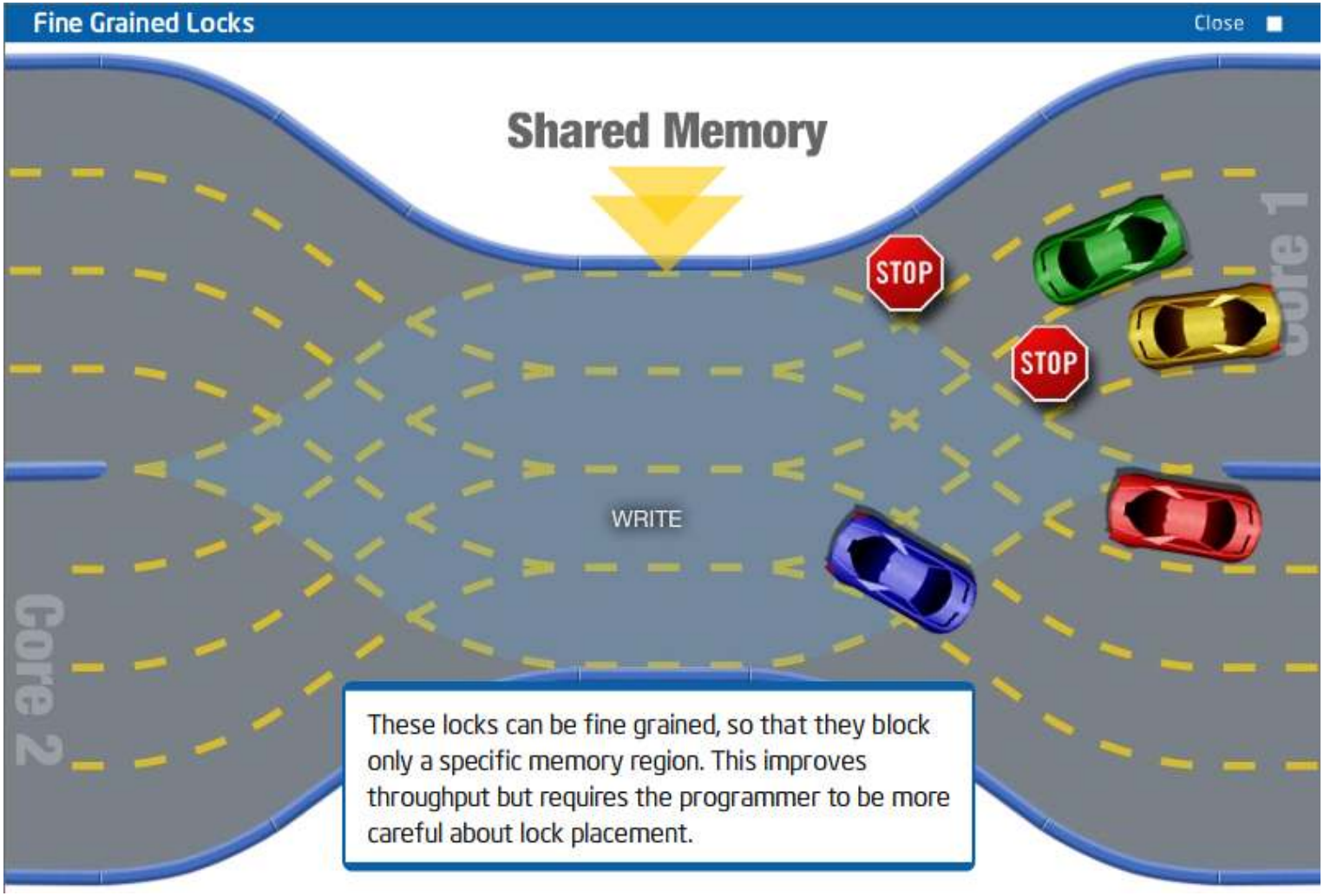
プログラム中で手動で LOCK を行った場合



インテル® C++ STM コンパイラーのコンセプト

VSUG
Visual Studio® User Group.

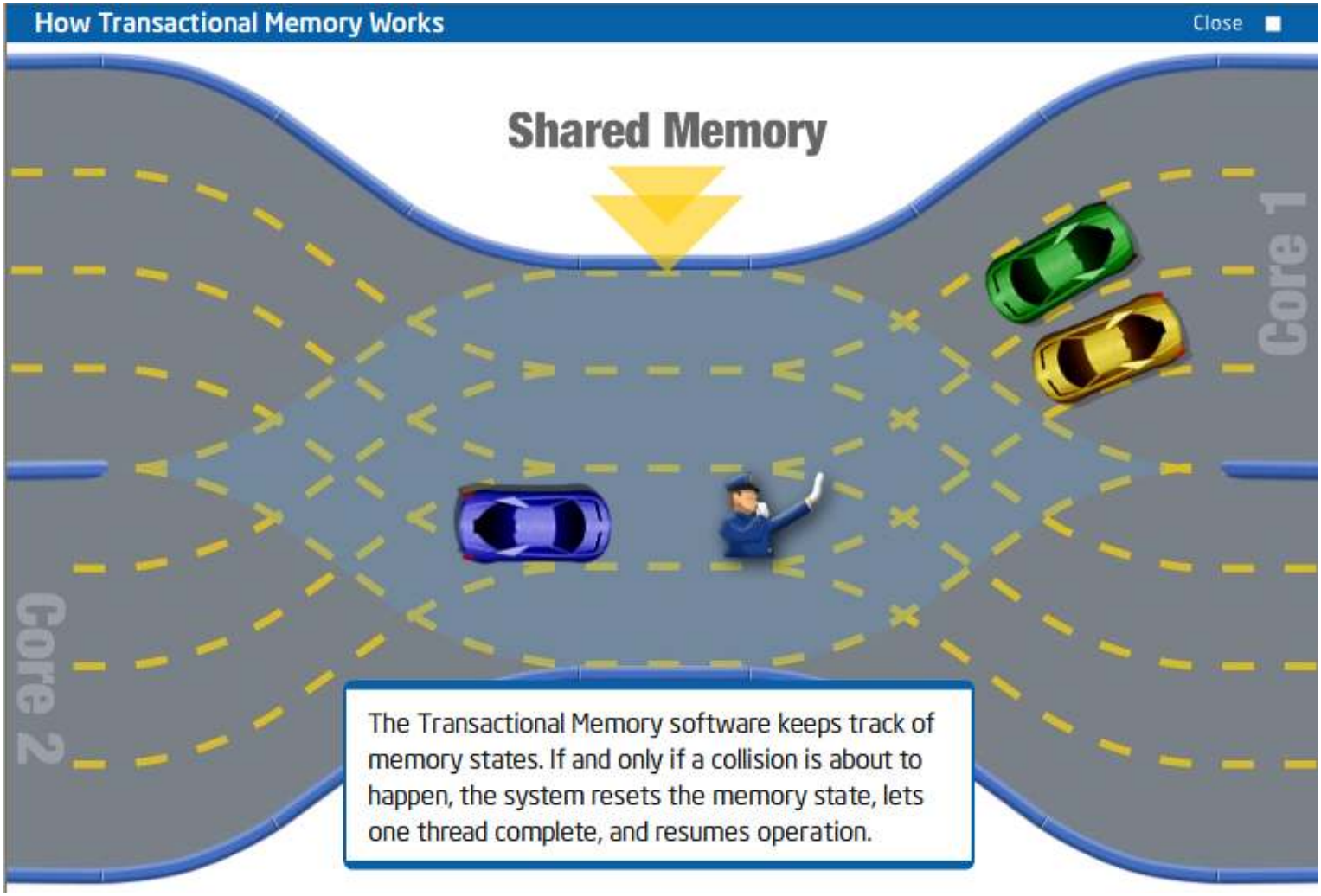
アクセスする領域ごとに LOCK を行った場合



インテル® C++ STM コンパイラーのコンセプト

VSUG
Visual Studio® User Group.

トランザクション・メモリー・ソフトウェアがアクセスを制御

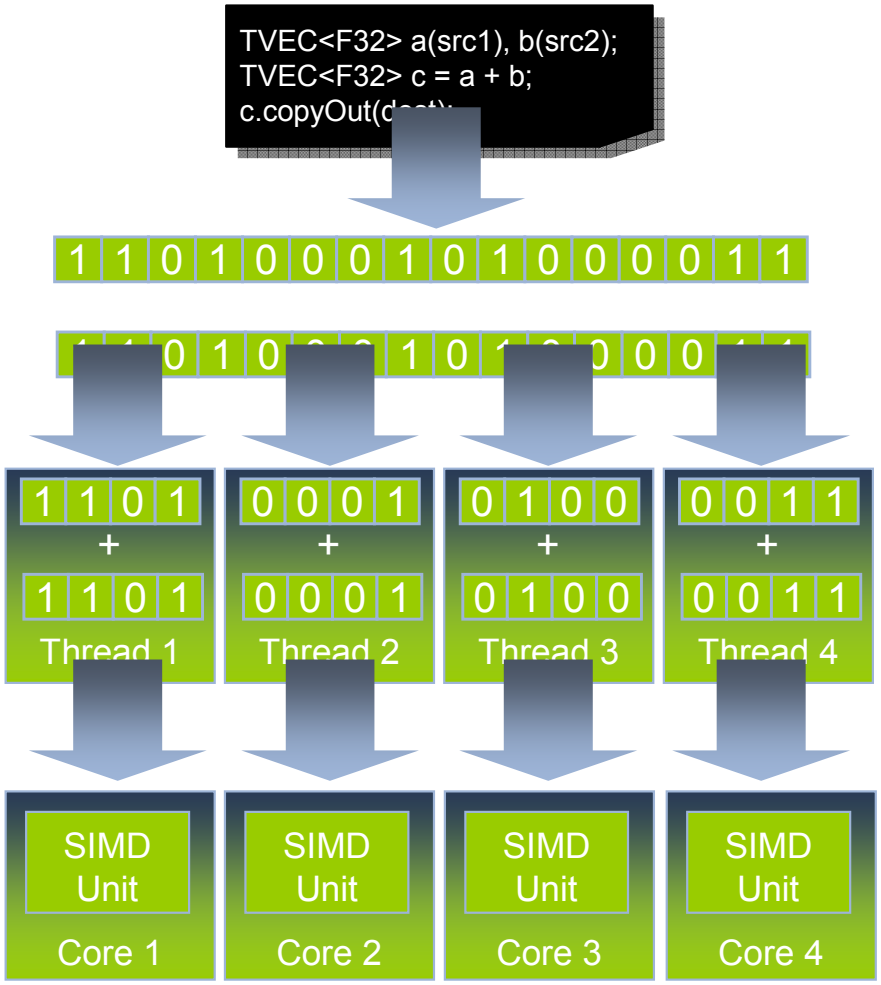


ラムダ関数

- **C++ 言語機能**
- **C++ 0x 標準を考慮**
- 引数としてコードを渡す
- **<>** オペレーター
- **TBB** における例:

```
…
  parallel_for(blocked_range<size_t>(0,to_scan.size(),100),
  <> ( const blocked_range<size_t>& r) const {
      for ( size_t i = r.begin(); i != r.end; i++) { … }     }
  MyTask(string &s, size_t *m, size_t *p) {…} //Init stuff
};
```

Ct: スループット・プログラミング・モデル



ユーザーはプロセッサー・コアに
依存しないC++コードを記述

Ct パラレル・ランタイム:
コアの増加に自動的にスケール

Ct JIT コンパイラー:
自動ベクトル化、SSE、
AVX、Larrabee

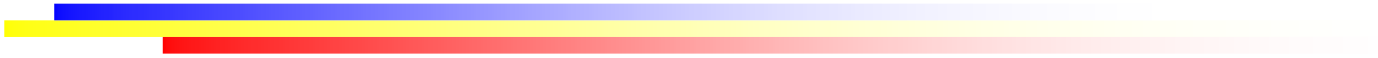
プログラマーはシリアルに考える; Ct が並列に展開

まとめと次のステップ

- ソフトウェア開発におけるマルチコアのインパクト
- コードをマルチコア対応にしてください
- 既存の API を進化させる
- 必要に応じて新たな API を開発する

どのように並列化を実装しますか？
ソフトウェア開発ツールが手助けできますか？

http://whatif.intel.com



VSUG
Visual Studio® User Group.

本日はありがとうございました


intel
Leap ahead™