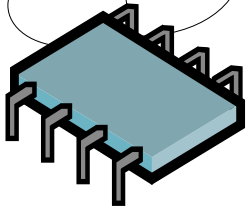


# 新しい並列for構文のご提案

2014/3/1

Boost.勉強会 #14 東京

**そのプログラム  
マルチコアを  
活用していますか？**



# 何の変哲もないプログラム

```
#include <vector>
using namespace std;

const int N = 1000000;
vector<MyData> data = /* N要素 */;

for (int i = 0; i < N; ++i) {
    process(data[i]);
}
```

# 何の変哲もないプログラム

forループの処理が重い…

```
vector<MyData> data = / "N要素" /,
```

```
for (int i = 0; i < N; ++i) {  
    process(data[i]);  
}
```

# 何の変哲もないプログラム

forループの処理が重い…  
そうだと、**並列処理**しよう

```
vector<MyData> data = /* N要素 */,
```

```
for (int i = 0; i < N; ++i) {  
    process(data[i]);  
}
```

# 並列処理



## マルチスレッド



### Pthreads, WinAPI, etc.

# 並列処理

え？

もう2014年ですよ？

Pthreads, WinAPI, etc.

# 並列処理



## マルチスレッド



## C++11 `std::thread`

or `Boost.Thread`



# 並列化 powered by C++11

```
int m = max(thread::hardware_concurrency(), 1u);
vector<thread> worker;
for (int i = 0; i < m; ++i) {
    worker.emplace_back([&](int id) {
        int r0 = N/m * id + min(N%m, id);
        int r1 = N/m * (id+1) + min(N%m, id+1);
        for (int j = r0; j < r1; ++j) {
            process(data[j]);
        }
    }, i);
}
for (auto& t : worker) t.join();
```

# 並列化 powered by C++11

```
int m = max(thread::hardware_concurrency(), 1u);
```

```
vector<thread> worker;
```

プロセッサ数取得

```
for (int i = 0; i < m; ++i) {
```

スレッド生成

```
    worker.emplace_back([&](int id) {
```

```
        int r0 = N/m * id + min(N%m, id);
```

```
        int r1 = N/m * (id+1) + min(N%m, id+1);
```

```
        for (int j = r0; j < r1; ++j) {
```

区間均等分割

```
            process(data[j]);
```

```
        }
```

```
    }, i);
```

forループ処理

```
}
```

```
for (auto& t : worker) t.join();
```

スレッド終了待機

# 並列化 powered by C++11

並列化のためのコードでなく  
**ロジック記述**に注力したい

```
int r1 = N/m * (id+1) + min(N%m, id+1);  
for (int j = r0; j < r1; ++j) {  
    process(data[j]);  
}  
}, i);  
}  
for (auto& t : worker) t.join();
```

# ループの並列化 といえば

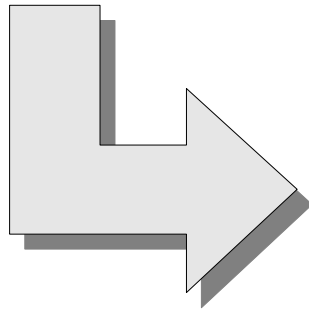
# ループの並列化 たとえば



# 並列化 powered by OpenMP

// 逐次処理

```
for (int i = 0; i < N; ++i) {  
    process(data[i]);  
}
```



// OpenMP並列化

```
#pragma omp parallel for  
for (int i = 0; i < N; ++i) {  
    process(data[i]);  
}
```

# 並列化 powered by OpenMP

// C++11標準ライブラリによる並列化

```
int m = max(thread::hardware_concurrency(), 1u);
vector<thread> worker;
for (int i = 0; i < m; ++i) {
    worker.emplace_back([&](int id) {
        int r0 = N/m * id + min(N%m, id);
        int r1 = N/m * (id+1) + min(N%m, id+1);
        for (int j = r0; j < r1; ++j) {
            process(data[j]);
        }
    }, i);
}
for (auto& t : worker) t.join();
```

// OpenMP並列化

#pragma omp parallel for

```
for (int i = 0; i < N; ++i) {
    process(data[i]);
}
```

どのように並列処理するか？

→ なにを並列処理するか？

手続的(HOW)な記述

→ 宣言的(WHAT)な記述

# おしまい





~~おいしい~~

**C++時代の**

# **新しい並列for構文のご提案**

2014/3/1

Boost.勉強会 #14 東京

# あらためまして

誰？

twitter @yohhoy / hatena id:yohhoy



何を？

C++と親和性の高いループ並列化技術の紹介

どうして？

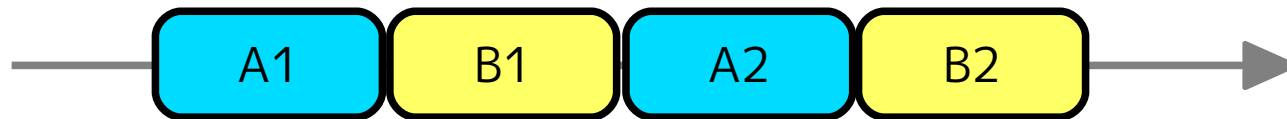
~~並列処理とか流行ってるから~~

C++11以降、並行・並列処理の検討が活発化

# 並行と並列

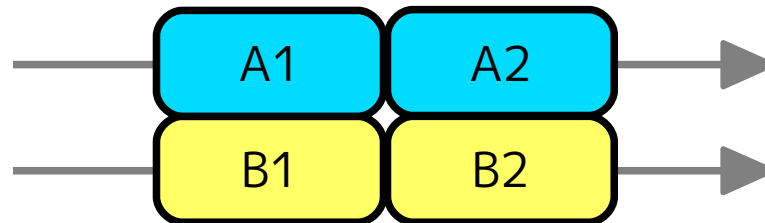
## 並行(Concurrent)

複数のタスクが存在し、それぞれが実行される。  
“本当に同時刻に処理されるかどうか”は気にしない。



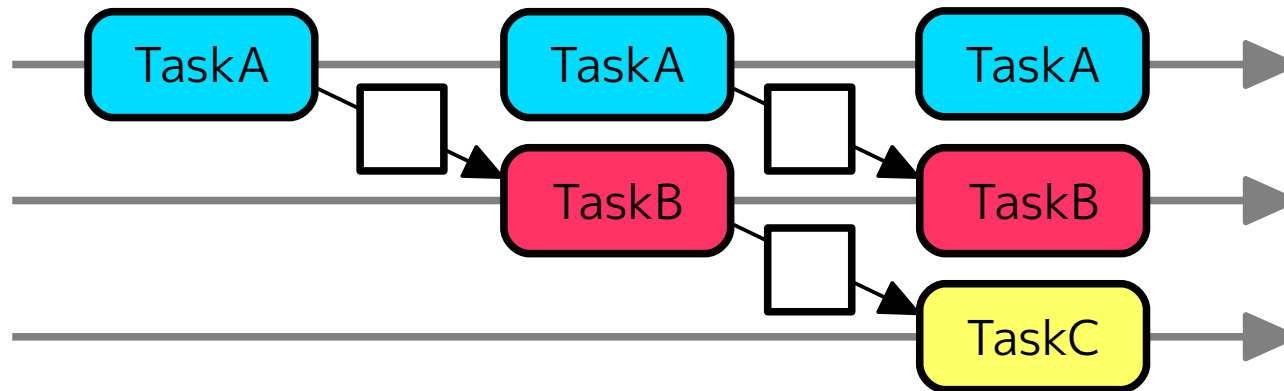
## 並列(Parallel)

高速化を目的として、複数H/Wを用いた同時計算を行う。  
“速く”ならなければ無価値。



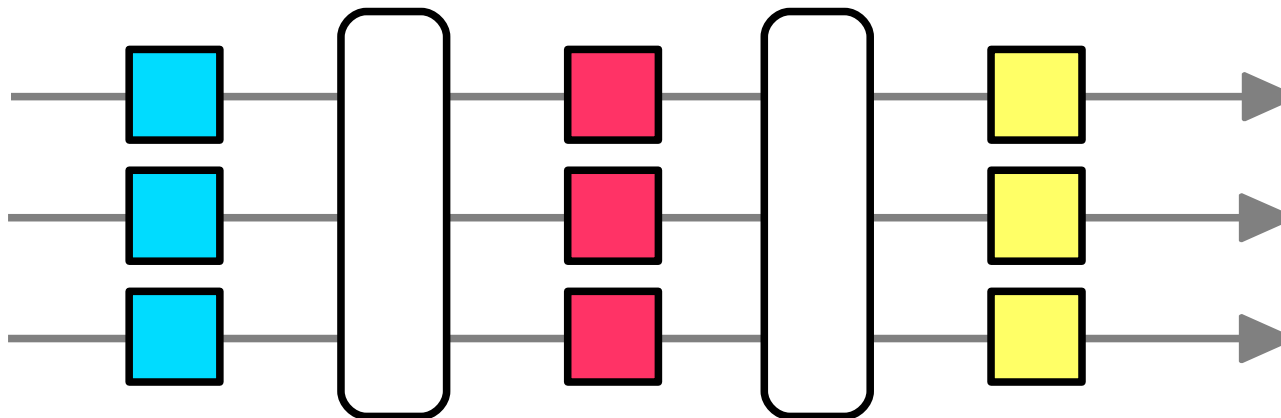
# 2種類の並列性

## タスク並列性(Task Parallelism)



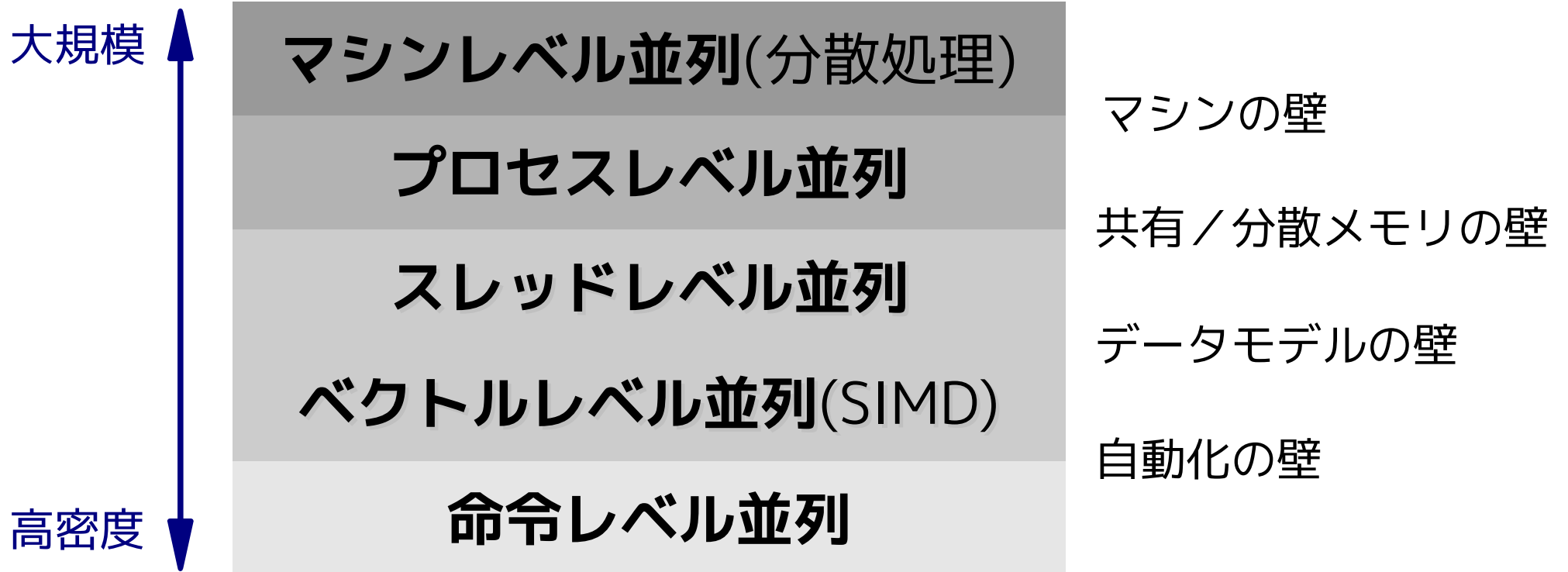
Ex) パイプライン

## データ並列性(Data Parallelism)




Ex) 並列ループ

# 並列化のレベル



(※今回の内容に合わせた独自の分類)



コンパイラによる  
完全な**自動並列化**は  
「**人類の夢**」

Fully automatic parallelization is still a

***“pipe dream”***

**【可算名詞】**

**〔アヘン吸引者が見るような〕**

**夢想、幻想**

*Parallel and Concurrent Programming in Haskell,  
O'REILLY, 2013*



# Real World Parallel Programming...

ライブラリ/言語拡張を利用した  
**並列処理プログラミング**が必要

マシンレベル並列(分散処理)

プロセスレベル並列

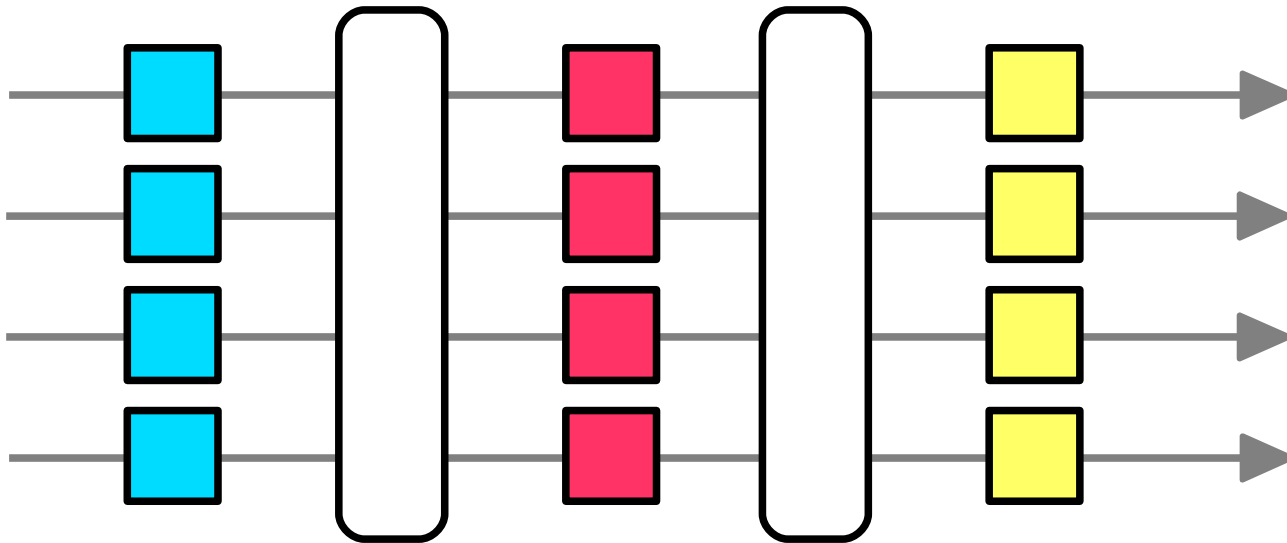
スレッドレベル並列

ベクトルレベル並列(SIMD)

命令レベル並列

# スレッドレベル並列

(マルチスレッド)



# スレッドレベル並列を支える技術

```
for (int i = 0; i < N; ++i) {  
    process(data[i]);  
}
```

ライブラリ

**TBB, PPL, libstdc++ ParallelMode**

言語拡張

**CilkPlus, OpenMP**

# OpenMP

```
#pragma omp parallel for  
for (int i = 0; i < N; ++i) {  
    process(data[i]); // C++例外をthrow...  
}
```

対応コンパイラ: GCC, Clang, MSVC, ICC...

# OpenMP

```
try {  
    #pragma omp parallel for  
    for (int i = 0; i < N; ++i) {  
        process(data[i]); // C++例外をthrow...  
    } // parallel regionここまで  
} catch (...) {  
    // 例外catch ??  
}
```



OpenMPスレッドをまたぐ  
C++例外送出手はNG

対応コンパイラ: GCC, Clang, MSVC, ICC...

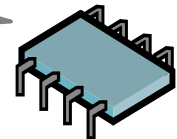
# OpenMPとC++例外

```
bool failed = false;
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    try {
        process(data[i]); // C++例外をthrow
    } catch (...) {
        // 同ースレッド内でcatch
        failed = true;
    }
} // parallel regionここまで
if (failed) {
    // ...
}
```

C++例外の扱いがめんどう  
(C++的でなく**ダサい**)

例外throwされても  
ループ中断できない

bug free?



# OpenMPとC++例外

```
bool failed = false;
#pragma omp parallel for reduction(||:failed)
for (int i = 0; i < N; ++i) {
    try {
        process(data[i]); // C++例外をthrow
    } catch (...) {
        // 同ースレッド内でcatch
        failed = true;
    }
} // parallel regionここまで
if (failed) {
    // ...
}
```

データ競合  
問題を修正

コード複雑化による  
厄介なバグの温床

# libstdc++ Parallel Mode拡張

```
// g++ source.cpp -D_GLIBCXX_PARALLEL -fopenmp
#include <algorithm>
std::for_each(begin(data), end(data),
[](Data& item) {
    process(item);
    // 例外送出手は禁止！
});
```



バックエンドはOpenMP  
C++例外送出手はNG

対応コンパイラ: GCC

対応OS: GCC+OpenMPが動く環境



# libstdc++ Parallel Mode拡張

```
// g++ source.cpp -fopenmp
#include <parallel/algorithm>
__gnu_parallel::for_each(begin(data), end(data),
[] (Data& item) {
    process(item);
    // 例外送出手は禁止！
});
```



バックエンドはOpenMP  
C++例外送出手はNG

対応コンパイラ: GCC

対応OS: GCC+OpenMPが動く環境

# Intel TBB

```
#include <tbb.h>
```

```
try {
```

```
    tbb::parallel_for_each(begin(data), end(data),
```

```
    [](Data& item) {
```

```
        process(item);
```

```
    });
```

```
} catch (...) {
```

```
    // 並列throwされた例外のうち最初の1つ
```

```
}
```

移植性に優れる

対応コンパイラ: GCC, Clang, MSVC, ICC...

対応OS: Linux, Windows, MacOS, (Android), (iOS)

# Microsoft ConcRT/PPL

```
#include <ppl.h>
try {
    concurrency::parallel_for_each(begin(data), end(data),
    [](Data& item) {
        process(item);
    });
} catch (...) {
    // 並列throwされた例外のうち最初の1つ
}
```

msvcrtにビルトイン

対応コンパイラ: MSVC

対応OS: Windows

# Intel CilkPlus

```
#include <cilk.h>
```

```
try {
```

```
    cilk_for (int i = 0; i < N; ++i) {
```

```
        process(data[i]);
```

```
    }
```

```
} catch (...) {
```

```
    // 並列throwされた例外のうち最初の1つ
```

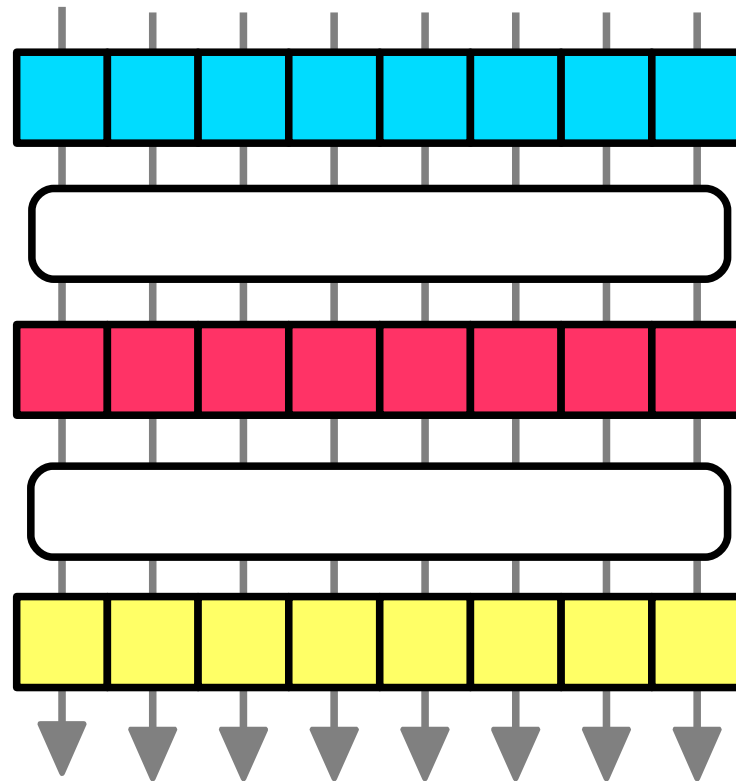
```
}
```

記述性に優れる

対応コンパイラ: ICC, (GCC), (Clang)

# ベクトルレベル並列

(SIMD, GPGPU, etc.)



# ベクトルレベル並列を支える技術

```
for (int i = 0; i < N; ++i) {  
    A[i] += B[i] * C[i];  
}
```

ライブラリ (+コンパイラサポート)

SIMD intrinsic, **Boost.SIMD**, **Thrust**, Bolt

言語拡張

**CilkPlus**, **C++AMP**, OpenMP 4.0, OpenACC

# SIMD intrinsic (組込み関数)

```
#include <xmmtrin.h> // x86/SSE
vector<float> A, B, C;
int i, n = N & ~3;
for (i = 0; i < n; i += 4) { // 4要素単位で処理
    __m128 a4 = _mm_loadu_ps(&A[i]);
    __m128 b4 = _mm_loadu_ps(&B[i]);
    __m128 c4 = _mm_loadu_ps(&C[i]);
    a4 = _mm_add_ps(a4, _mm_mul_ps(b4, c4));
    _mm_storeu_ps(&A[i], a4);
}
for (; i < A.size(); ++i) // 端数処理
    A[i] += B[i] * C[i];
```

要 アセンブラ知識  
(どんな命令がある?)



命令セット毎に  
異なるコードが必要  
(SSE2, AVX, NEON...)

対応コンパイラ: GCC, Clang, MSVC, ICC...

# Boost.SIMD (NT<sup>2</sup>)

```
#include <boost/simd/***.hpp>
vector<float, boost::simd::allocator<float>> A, B, C;
typedef boost::simd::pack<float> FVec;
int step = FVec::static_size;
int i, n = (N / step) * step;
for (i = 0; i < n; i += step) { // 処理単位は命令セット依存
    FVec a = boost::simd::aligned_load<FVec>(&A[i]);
    FVec b = boost::simd::aligned_load<FVec>(&B[i]);
    FVec c = boost::simd::aligned_load<FVec>(&C[i]);
    a += b * c;
    boost::simd::aligned_store(a, &A[i]);
}
for (; i < A.size(); ++i) // 端数処理
    A[i] += B[i] * C[i];
```

SIMD処理を一般化  
(x86/SSE系列, AVXと  
PPC/Altivecに対応)

ARM/NEON未対応

対応コンパイラ: GCC, Clang, MSVC, ICC



# Intel CilkPlus / Array Notation

```
vector<float> A, B, C;
```

```
float* a = &A[0];
```

```
float* b = &B[0];
```

```
float* c = &C[0];
```

宣言的なコーディング  
(ベクトル演算のセマンティクス)

```
a[:N] = a[:N] + b[:N] * c[:N];
```

対応コンパイラ: ICC, (GCC), (~~Clang~~)

Array Notation未対応

# Thrust (GPGPU; CUDA)

CPU向け(TBB, OpenMP)  
バックエンドも提供

```
#include <thrust/***.h>
thrust::device_vector<float> A, B, C;
struct fma_functor {
    template <typename Tuple>
    __device__ void operator()(Tuple t) {
        thrust::get<0>(t) += thrust::get<1>(t) * thrust::get<2>(t);
    } // Tuple = [A, B, C]
};
thrust::transform(
    thrust::make_zip_iterator(thrust::make_tuple(begin(A), begin(B), begin(C))),
    thrust::make_zip_iterator(thrust::make_tuple(end(A), end(B), end(C))),
    fma_functor() );
```

対応コンパイラ: CUDA Compiler, GCC, Clang, MSVC

# Microsoft C++ AMP (GPGPU; DirectX)

```
#include <amp.h>
vector<float> A, B, C;
concurrency::array_view<float, 1> a(N, &A[0]);
concurrency::array_view<float, 1> b(N, &B[0]);
concurrency::array_view<float, 1> c(N, &C[0]);
concurrency::parallel_for_each(
    a.extent,
    [=](concurrency::index<1> idx) restrict(amp) {
        a[idx] += b[idx] * c[idx];
    }
);
a.synchronize(); // GPU→CPUデータ同期
```

CPU/GPU処理を  
統一的に記述

対応コンパイラ: MSVC, (Clang)

# スレッド/ベクトル 並列化技術

	ライブラリ	言語拡張
スレッド レベル並列	<b>TBB</b> <b>PPL</b> libstdc++ ParallelMode	<b>CilkPlus</b> OpenMP
ベクトル レベル並列	<b>Boost.SIMD</b> <b>Thrust</b> SIMD intrinsic	<b>C++AMP</b> <b>CilkPlus</b>

**スレッドレベル並列**

**VS.**

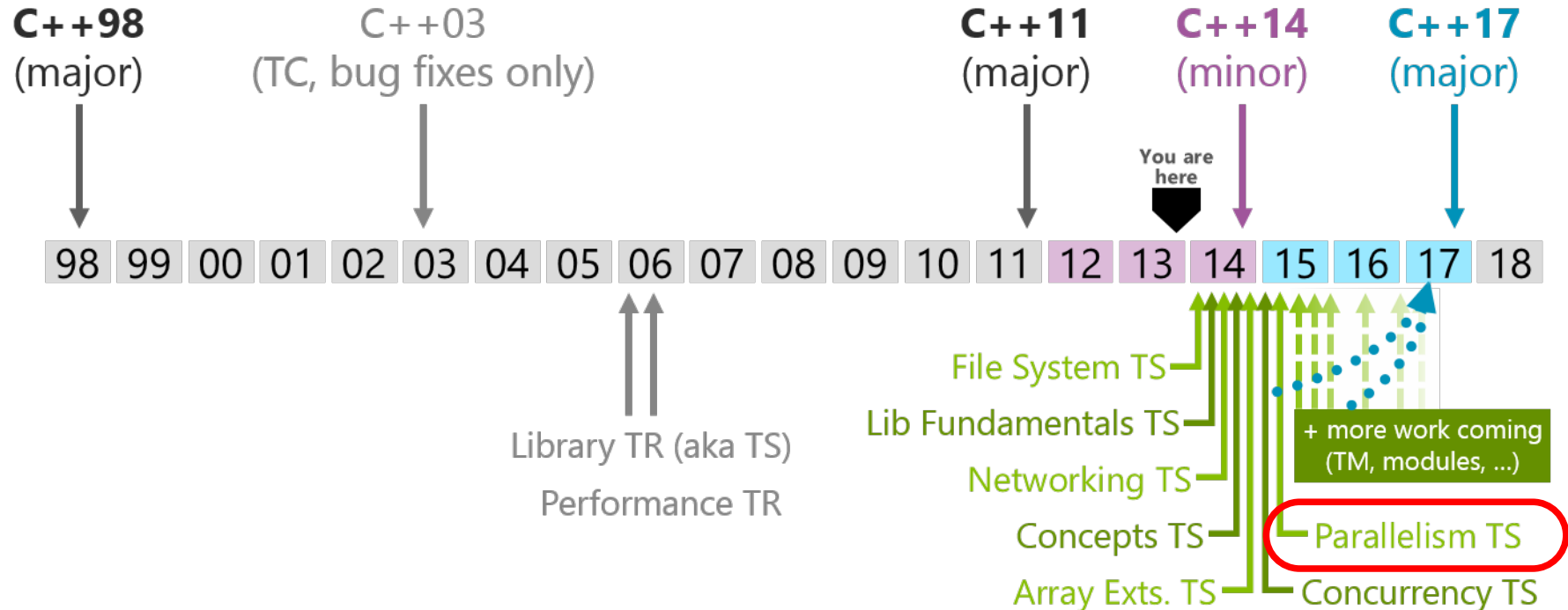
**ベクトルレベル並列**



スレッドレベル並列  
+  
ベクトルレベル並列  
=  
C++1y ParallelTS

# C++ 並列擴張

N3850 “Working Draft, Technical Specification for C++ Extensions for Parallelism”



<http://isocpp.org/blog/2013/10/trip-report-fall-iso-c-meeting>

# C++並列拡張

N3850 “Working Draft, Technical Specification for C++ Extensions for Parallelism”

**並列アルゴリズム**のインターフェースを規定

C++標準アルゴリズムを並列実行可能に**拡張**

(`<algorithm>`, `<numeric>`, `<memory>`ヘッダ)

API設計は **Thrust** をベースとする



# C++並列拡張

N3850 “Working Draft, Technical Specification for C++ Extensions for Parallelism”

C++並列ライブラリの広範な実装経験確立が目的

検討用実装：<https://github.com/n3554/n3554/>

今は `std::experimental::parallel` 名前空間

→ `std` 名前空間 への昇格を目指す

# C++1y ParallelTS (n3850)

```
#include <algorithm>
```

```
using namespace std;
```

```
try {
```

```
    for_each(
```

```
        begin(data), end(data), [](Data& item) {
```

```
            process(item);
```

```
        });
```

```
} catch (...) {
```

```
    // ...
```

```
}
```

# C++1y ParallelTS (n3850)

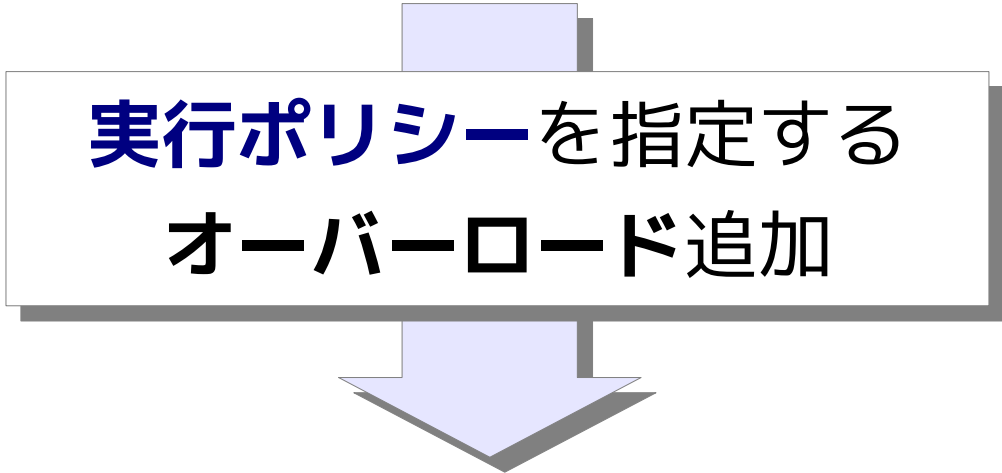
```
#include <experimental/algorithm>
#include <experimental/execution_policy>
using namespace std::experimental::parallel;
try {
    for_each(
        par, // 並列実行ポリシー
        begin(data), end(data), [](Data& item) {
            process(item);
        });
} catch (exception_list& exlist) {
    for (exception_ptr ex : exlist) {
        // 並列throwされた全ての例外が捕捉される
    }
}
```

# C++1y ParallelTS (n3850)

// C++標準アルゴリズム

```
template <class In, class F>
```

```
In for_each( In first, In last, F f );
```



実行ポリシーを指定する  
オーバーロード追加

// C++1y ParallelTS

```
template <class ExecPolicy, class In, class F>
```

```
In for_each( ExecPolicy&& exec, In first, In last, F f );
```

# C++1y ParallelTS (n3850)

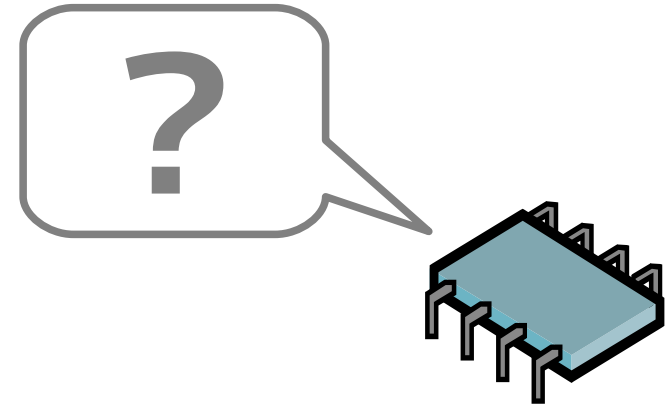
<b>&lt;algorithm&gt;</b>	transform	stable_partition	<b>&lt;numeric&gt;</b>
all_of	replace	partition_copy	<i>reduce</i>
none_of	replace_if	partition_point	inner_product
for_each	replace_copy	sort	<i>exclusive_scan</i>
<i>for_each_n</i>	replace_copy_if	stable_sort	<i>inclusive_scan</i>
find	fill	partial_sort	adjacent_difference
find_if	fill_n	partial_sort_copy	
find_if_not	generate	is_sorted	<b>&lt;memory&gt;</b>
find_end	generate_n	is_sorted_until	uninitialized_copy
find_first_of	remove	nth_element	uninitialized_copy_n
adjacent_find	remove_if	merge	uninitialized_fill
count	remove_copy	inplace_merge	uninitialized_fill_n
count_if	remove_copy_if	includes	
mismatch	unique	set_union	
equal	unique_copy	set_intersection	
search	reverse	set_difference	
search_n	reverse_copy	set_symmetric_difference	
copy	rotate	min_element	
copy_n	rotate_copy	max_element	
move	is_partitioned	minmax_element	
swap_range	partition	lexicographical_compare	

一部は明示的に除外  
<algorithm> shuffle  
<numeric> accumulate  
etc.

# 実行ポリシー

## seq 逐次実行

呼び出しスレッド上で実行



## par 並列実行

複数スレッド上で順序付けされない実行(*unordered*) もしくは  
1スレッド上で非決定順の実行(*indeterminately sequenced*)

## vec ベクトル実行

複数スレッド上で順序付けされない実行(*unordered*) もしくは  
1スレッド上で序列化されない実行(*unsequenced*)

# 実行ポリシー

## seq 逐次実行

シングルスレッド実行（従来C++標準アルゴリズムと同一）

## par 並列実行

マルチスレッドによる並列実行

または シングルスレッド実行（実行順序は同不順）

## vec ベクトル実行

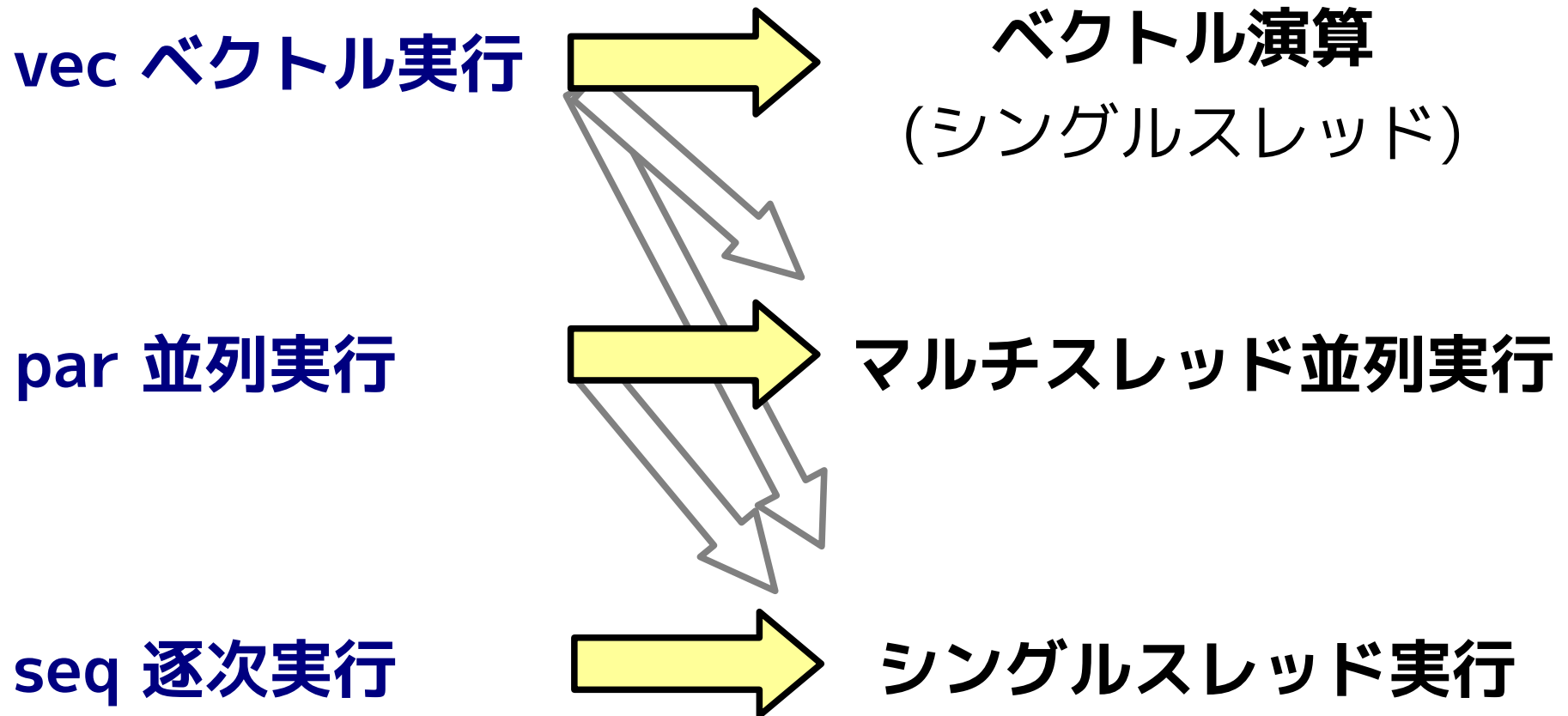
1スレッド上でベクトル並列実行

または マルチスレッドによる並列実行

または シングルスレッド実行（実行順序は同不順）

# 実行ポリシーと実行形態

コンパイラやライブラリが実行形態を選ぶ





# 実行ポリシー / static

コンパイル時に実行ポリシーを指定

```
vector<float> data = ...;
```

```
// 逐次ソート
```

```
sort( seq, begin(data), end(data) );
```

```
// sort( begin(data), end(data) ); と等価
```

```
// マルチスレッドによる並列ソート
```

```
sort( par, begin(data), end(data) );
```

```
// ベクトル演算器によるソート
```

```
sort( vec, begin(data), end(data) );
```

# 実行ポリシー / dynamic

ランタイムに実行ポリシーを選択

```
vector<float> data = ...;
// デフォルトは逐次ソート
execution_policy exec = seq;
if ( data.size() > threshold ) {
    // データ数が多いければ並列ソートに切替
    exec = par;
}
sort( exec, begin(data), end(data) );
```

# C++1y ParallelTS (n3850)

```
#include <experimental/algorithm>
#include <experimental/execution_policy>
using namespace std::experimental::parallel;
try {
    for_each(
        par, // 並列実行ポリシー
        begin(data), end(data), [](Data& item) {
            process(item);
        });
} catch (exception_list& exlist) {
    for (exception_ptr ex : exlist) {
        // 並列throwされた全ての例外が捕捉される
    }
}
```

# exception\_list

並列アルゴリズムからthrowされうる例外型

並列にthrowされた全ての例外オブジェクトを格納

```
class exception_list : public exception
{
public:
    size_t size() const;
    iterator begin() const;
    iterator end() const;
private:
    list<exception_ptr> exceptions_; // exposition only
};
```

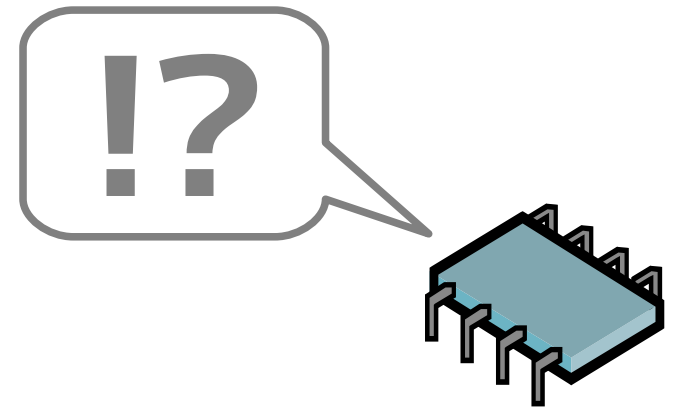
# 実行ポリシーとC++例外

## seq 逐次実行 と par 並列実行

関数オブジェクトからの例外は、`exception_list`型に補足されて呼び出し元へthrowされる。

## vec ベクトル実行

関数オブジェクトから例外throwされた場合、`std::terminate`が呼び出されてプログラム終了する。



# 実行ポリシーとC++例外

## seq 逐次実行 と par 並列実行

任意のスレッド上で実行されうるセマンティクス  
複数スレッドからthrowされた例外を全て補足

## vec ベクトル実行

ベクトル並列化(SIMD, GPGPUなど)可能なセマンティクス  
コンパイラによるベクトル化を阻害しないよう例外使用禁止

# 実行ポリシーと制約事項

	並列性の セマンティクス	処理 順序	例外 throw	関数オブジェクト 制約事項
<b>vec</b> ベクトル実行	ベクトル演算 (シングルスレッド 上の並列処理)	順不同	✕	相互干渉は全てNG
<b>par</b> 並列実行	マルチスレッド 並列処理	順不同	○	順序依存性はNG (Ex. 他反復の待機処理 でDeadLock発生)
<b>seq</b> 逐次実行	シングルスレッド 逐次処理	安定	○	特になし 標準アルゴリズムと同じ



# C++1y ParallelTS

TS(Technical Specification)はC++標準ではありません  
しかも**ParallelTS**はまだWorking Draft段階です  
→ 今後もしろいろ仕様変更されます

参考までに：

C++ Transactional Memory拡張 (@Boost.勉強会#10)

<http://www.slideshare.net/yohhoy/boostjp10-tm-20120728>

- 紹介当時(2012/7)と現在で大きく様変わりしましたホントに
- 基本コンセプトの名称変更・セマンティクス修正、  
キーワード・属性の全面変更、etc.



# Need More Speed?

速度性能はコンパイラやライブラリ実装に依存します

ただし、一般論として…

**CPU bound** (I/O boundは並行処理向け)

並列化可能な**アルゴリズム**への**変更**

タスク間の**依存性低減** (同期箇所・共有データの削減)

データ**メモリ配置** (局所性、偽共有、アライメント)

Locality

False sharing

Alignment

**計測！計測！計測！**

# まとめ

**宣言的**なコード記述による**並列ループ**構文  
**スレッドレベル並列の技術**

**TBB, PPL, CilkPlus**, OpenMP, etc.

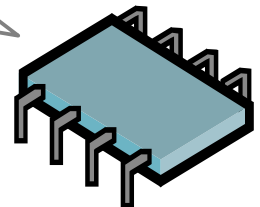
**ベクトルレベル並列の技術**

**Boost.SIMD, Thrust, CilkPlus, C++AMP**,

OpenMP 4.0, OpenACC, Bolt, etc.

**ParallelTS**では**スレッド/ベクトル並列統合**を目指す?

ライブラリ実装とコンパイラ技術  
今後の発展にご期待ください



# 参考URL(1)

**n3850 “Working Draft, Technical Specification for C++ Extensions for Parallelism”** (Thrustベース 並列アルゴリズム提案)

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3850.pdf>

<https://github.com/n3554/n3554/>

**n3571 “A Proposal to add Single Instruction Multiple Data Computation to the Standard Library”** (Boost.SIMDベース データ型+操作関数提案)

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3571.pdf>

**n3831 “Language Extensions for Vector level parallelism”**

(CilkPlusベース ベクトル並列ループ構文提案)

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3831.pdf>

**n3851 “Multidimensional bounds, index and array\_view”**

(C++AMPベース データ型提案)

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3851.pdf>

# 参考URL(2)

**OpenMP** <http://openmp.org/>

**Thrust** <http://thrust.github.io/>

## **Intel TBB(Threading Building Blocks)**

<https://www.threadingbuildingblocks.org/>

## **Microsoft ConCRT(Concurrency Runtime) / PPL(Parallel Pattern Library)**

<http://msdn.microsoft.com/ja-jp/library/dd492418.aspx>

## **libstdc++ Parallel Mode**

[http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel\\_mode.html](http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html)

**Intel CilkPlus** <https://www.cilkplus.org/>

**GCC実装** <https://www.cilkplus.org/build-gcc-cilkplus>

**Clang実装** <http://cilkplus.github.io/>

## **Microsoft C++ AMP(Accelerated Massive Parallelism)**

<http://msdn.microsoft.com/ja-jp/library/hh265137.aspx>

**Clang** <http://isocpp.org/blog/2012/12/shevlin-park>

**Boost.SIMD** <https://github.com/MetaScale/nt2/>

## **SIMD intrinsic function**

**x86** <http://software.intel.com/sites/landingpage/IntrinsicsGuide/>

**ARM** <http://infocenter.arm.com/help/topic/com.arm.doc.dui0491c/CIHJBEFE.html>

# [予備] OpenMPでC++例外伝播

```
#include <omp.h>
try {
    std::exception_ptr ep;
    omp_lock_t ep_guard;
    omp_init_lock(&ep_guard);
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        try {
            process(data[i]); // C++例外をthrow
        } catch (...) {
            omp_set_lock(&ep_guard);
            if (!ep) ep = std::current_exception(); // 最初の例外のみ保存
            omp_unset_lock(&ep_guard);
        }
    }
    omp_destroy_lock(&ep_guard);
    if (ep) std::rethrow_exception(ep); // 例外をparallel region外で再throw
} catch (...) {
    // 例外処理
}
```

# [予備] OpenMP vs. CilkPlus/TBB/PPL

## CilkPlus, TBB, PPL

スレッドプール + Work Stealing

実行時に論理タスク → 物理スレッドへのマッピング

高いスケールラビリティを得やすい

## OpenMP

parallel指示文で明示的にスレッド生成（≠宣言的）

スレッドプールによる実装も存在（処理系の品質）

OpenMP 3.0で“task”導入（※MSVCは非対応）

# [予備] OpenMP 4.0, n3831

// 逐次処理

```
for (int i = 0; i < N; ++i) {  
    A[i] += B[i] * C[i];  
}
```

// OpenMP 4.0 / SIMD construct

**#pragma omp simd aligned(A,B,C:16)**

```
for (int i = 0; i < N; ++i) {  
    A[i] += B[i] * C[i];  
}
```

// n3831 Language Extensions

// for Vector level parallelism

```
for simd (int i = 0; i < N; ++i) {  
    A[i] += B[i] * C[i];  
}
```