

インテル[®] Cilk[™] Plus ユーザーガイド

本書はインテル[®] Parallel Studio 2011 に含まれる、インテル[®] C/C++ コンパイラーの「Intel[®] C++ Compiler 12.0 User and Reference Guides」の「Using Intel[®] Cilk[™] Plus」章を翻訳加筆したものです。

October 12, 2010!
翻訳 Powered by SUGATeCH

目次

はじめに	3
インテル® Cilk™ プログラムのビルド、実行およびデバッグ	11
インテル® Cilk™ Plus 言語機能	14
インテル® Cilk™ Plus の実行モデル	23
レデューサー	30
オペレーティング・システムに関する考察	44
インテル® Cilk™ Plus ランタイム API.....	47
競合状態を理解する.....	49
ロックを利用する	54
インテル® Cilk™ Plus の性能上の考察	58
追加情報	62
用語	63

はじめに

インテル® Cilk™ Plus はインテル® C++ コンパイラーに含まれる機能です。インテル® Cilk™ Plus を導入することにより、新規に開発するプログラムや既存のプログラムに並列性を実装しプログラムのパフォーマンスを改善します。このリリースでは、次の環境で IA-32 もしくは Intel® 64 アーキテクチャー・ベースのプログラムのビルドをサポートします；

32 ビットおよび 64 ビット Microsoft Windows* オペレーティング・システム
32 ビットおよび 64 ビット Linux* オペレーティング・システム

このドキュメントに記載されるほとんどの情報はすべてのプラットフォームに共通しますが、プラットフォーム固有の情報は個別に記載されています。

このドキュメントの対象読者

C もしくは C++ プログラミングの知識を持ち、インテル® Cilk™ Plus の導入を検討する開発者を対象読者としていますが、C や C++ の専門知識があれば最適です。

必要要件

システム要件とインストール手順の詳細は、製品に含まれるリリースノートを参照してください。

コンパイラーとツールの使い方に馴染むため、インテル® Parallel Composer 2011 やインテル® Parallel Composer XE 2011 をインストールして、コンパイラーに含まれるサンプル・プログラムをビルドして実行してみましょう。

補足情報

巻末の追加情報には、関連する追加情報が含まれています。また、用語集で解説される単語は太字で記載されます。

利用する前に

インテル® Cilk™ Plus は、複数のプロセッサを効率よく利用するため C と C++ プログラムに細粒度のタスクを実装し、簡単に(既存もしくは新規の)プログラムに並列性を導入できます。

インテル® Cilk™ Plus は、特に「分割統治(Divide and Conquer)」アルゴリズムに適したツールです。分割統治は問題を小さなサブ・プログラム(タスク)に分割し、それらが独立して問題を処理し結果を結合します。再帰関数は分割統治アルゴリズムにしばしば利用され、多くの場合良い結果をもたらします。

タスクは独立した関数やループの繰り返しに実装することができ、キーワードにより並列実行できる関数呼び出しやループを指示します。

以降の節では、「ワーカー」という用語は、オペレーティング・システムのスレッドがプログラムにおけるタスクを実行する、という意味で利用されます。

インテル® Cilk™ Plus サンプル・プログラムのビルドと実行

リリースノートで説明されるように、サンプルコードは名前付けされた各ホルダーの下に置かれます。ここでは、qsort のサンプルを使用する場合を例に説明します。

先へ進む前に、2つ以上のプロセッサ・コアを搭載するシステムに、製品(インテル® Parallel Composer 2011 もしくは Intel® C++ Composer XE 2011)がインストールされていることを確認してください。シングルコアのシステムでは、サンプルをビルドして実行することはできますが、性能向上を評価することはできません。

qsort をビルドする

ビルドオプションの詳しい使い方は「インテル® Cilk™ Plus プログラムのビルド、実行およびデバッグ」の節で説明します。ここではデフォルト設定を利用します。

インテル® C++ コンパイラーのコマンドラインからの起動名は、Windows* システムでは icl、Linux システムでは icc です。

Linux* システムの場合；

1. qsort サンプルコードのディレクトリーで移動 (cd INSTALLDIR/examples/qsort)
2. make コマンドを実行する。make が失敗した場合、INSTALL/bin ディレクトリーからインテル® Cilk™ Plus 関連のPATH 環境変数が設定されていることを確認してください。
3. カレント・ディレクトリーに実行可能な qsort がビルドされます。

Windows* システムの場合；

1. Microsoft Visual Studio* を利用する場合：ソリューション(qsort.sln)を開き、Release バージョンをビルドします。実行ファイルは EXDIR¥qsort¥Release¥qsort.exe に作成されます。

ビルドエラーが発生した場合、Visual Studio 環境でインテル® C++ コンパイラーを利用するようになっているか確認してください。プロジェクト名を右クリックし、次の何れかを選択します：

Intel Parallel Composer 2011 -> Use Intel C++

Intel C++ Composer XE 2011 -> Use Intel C++

2. icl を利用してコマンドラインからビルドする場合：icl qsort.cpp
コンパイルが失敗した場合、インテル® Cilk™ Plus 関連のPATH 環境変数が設定されていることを確認してください。

qsort を実行する

最初に qsort が正しく実行されることを確認します。引数なしで qsort を起動した場合、10,000,000 個の整数配列を生成しソートします。次に例を示します。

```
>qsort
Sorting 10000000 integers
5.641 seconds
Sort succeeded.
```

マルチコア・システムで性能向上を確認します

インテル® Cilk™ Plus プログラムは、オペレーティング・システムにプロセッサ・コア数を問い合わせ、デフォルトで利用可能な最大のプロセッサ・コアを利用します。CILK_NWORKER 環境変数を利用して、ワーカーの個数を制御することができます。

次の例は、8 コア・システムでの実行結果です。性能向上はアプリケーションの並列性とコア数に依存することが判ります。

Linux* OS:

```
>CILK_NWORKERS=1 qsort
Sorting 10000000 integers
2.909 seconds
Sort succeeded.

>CILK_NWORKERS=2 qsort
Sorting 10000000 integers
1.468 seconds
Sort succeeded.

>CILK_NWORKERS=4 qsort
Sorting 10000000 integers
0.798 seconds
Sort succeeded.

>CILK_NWORKERS=8 qsort
Sorting 10000000 integers
0.438 seconds
Sort succeeded.
```

Windows* コマンドライン:

```
>set CILK_NWORKERS=1
>qsort
Sorting 10000000 integers
909 seconds
Sort succeeded.

>set CILK_NWORKERS=2
>qsort
Sorting 10000000 integers
Sort succeeded.
```

```
>set CILK_NWORKERS=4
>qsort
Sorting 10000000 integers
798 seconds
Sort succeeded.

>set CILK_NWORKERS=8
>qsort
Sorting 10000000 integers
438 seconds
Sort succeeded.
```

Microsoft Visual Studio* 環境でワーカー数を変更するには:

コンテキスト・メニューからプロジェクトを右クリックしプロパティーを開きます。

設定のカテゴリーで Intel Debugging を選択し、Number of Intel® Cilk™ Plus Threads プロパティーにワーカー数を設定します。

C++ プログラムを変換する

ここでは、インテル® Cilk™ Plus を使用して並列プログラムを作成する手順を説明します:

1. 通常この作業は、C++ で記述された並列化の対象となる基本関数やアルゴリズムを実装するシリアルプログラムを対象として開始します。並列化を実装する前に、シリアルプログラムが正しく動作することを確認してください。シリアルプログラム中のバグは並列プログラムでも発生し、それらを発見して修正するのはいっそう困難になります。
2. 並列化による効果があるプログラム領域を特定します。比較的長い時間実行され、独立して処理できるコード領域が候補となります。
3. 並列に実行するため次の 3 つのキーワードを利用します。

`_Cilk_spawn` (プログラムで `<cilk/cilk.h>` をインクルードしている場合は `cilk_spawn` が利用できます)は、呼び出し元(親)と並列に実行できる関数(子)呼び出しを指示します。

`_Cilk_sync` (プログラムで `<cilk/cilk.h>` をインクルードしている場合は `cilk_sync` が利用できます)は、スポンされたすべての子が完了するのを待機することを指示します。

`_Cilk_for` (プログラムで `<cilk/cilk.h>` をインクルードしている場合は `cilk_for` が利用できます)は、ループのすべての反復が同時に実行できることを指示します。

4. プログラムをビルドします:

Windows* OS: `icl` コマンドライン・ツールか Microsoft Visual Studio* 環境でコンパイルを行います。Visual Studio* を利用する場合、プロジェクトのコンテキスト・メニューで Use Intel C++ によってコンパイラが切り替えられていることを確認します。

Linux* OS: `icc` コマンドを利用します。

5. プログラムを実行します。並列化したプログラムにデータ競合が発生していなければ、結果はシリアルプログラムと同じとなります。
6. レデューサー(reducer)やロックを利用するか、もしくは競合が発生しないようにコードを変更し、問題を解決します。

sort プログラムの例題を利用して、上記の手順を検証してみます。

シリアルプログラム

以降のリストは Quicksort を並列化するために、インテル® Cilk™ Plus でどのように記述するかを示しています。

次のリストでは、標準 C ライブラリーの qsort 関数と区別するため、関数名を sample_qsort としています。また、リスト中のいくつかの行は削除されていますが、オリジナルの行番号をそのまま使用しています。

```

9 #include<algorithm>
10
11 #include <iostream>
12 #include <iterator>
13 #include <functional>
14
15 // Sort the range between begin and end.
16 // "end" is one past the final element in the range.
19 // This is pure C++ code before Cilk conversion.
20
21 void sample_qsort(int * begin, int * end)
22 {
23     if (begin != end) {
24         --end; // Exclude last element (pivot)
25         int * middle = std::partition(begin, end,
26             std::bind2nd(std::less<int>(),*end));
27
28         std::swap(*end,*middle); // pivot to middle
29         sample_qsort(begin, middle);
30         sample_qsort(++middle, ++end); // Exclude pivot
31     }
32 }
33
34 // A simple test harness
35 int qmain(int n)
36 {
37     int *a = new int[n];
38
39     for (int i = 0; i < n; ++i)
40         a[i] = i;
41
42     std::random_shuffle(a, a + n);
43     std::cout << "Sorting " << n << " integers"
44         << std::endl;
45     sample_qsort(a, a + n);

```

```

48
49 // Confirm that a is sorted and that each element
    // contains the index.
50 for (int i = 0; i < n-1; ++i) {
51     if ( a[i] >= a[i+1] || a[i] != i ) {
52         std::cout << "Sort failed at location i="
                    << i << " a[i] = "
53                 << a[i] << " a[i+1] = " << a[i+1]
                    << std::endl;
54         delete[] a;
55         return 1;
56     }
57 }
58 std::cout << "Sort succeeded." << std::endl;
59 delete[] a;
60 return 0;
61 }
62
63 int main(int argc, char* argv[])
64 {
65     int n = 10*1000*1000;
66     if (argc > 1)
67         n = std::atoi(argv[1]);
68
69     return qmain(n);
70 }

```

`_Cilk_spawn` を使用して並列化する

`_Cilk_spawn` キーワードは、キーワードで示される関数(子)とその後に続くステートメント(親)が同時に実行されることを表します。キーワードは並列化を指示しますが、必ずしも並列操作が必要なわけではありません。インテル® Cilk™ Plus は、マルチプロセッサ(マルチコア・プロセッサ)が利用可能な時、どの操作が並列実行可能か動的に判断します。`_Cilk_sync` ステートメントは、先行するすべての `_Cilk_spawn` された関数が完了するまで、指示された場所以降を実行しないで待機することを指示します。`_Cilk_sync` は、他の関数内でスポーンされた並列処理には影響しません。

```

19 #include <cilk/cilk.h>

21 void sample_qsort(int * begin, int * end)
22 {
23     if (begin != end) {
24         --end; // Exclude last element (pivot)
25         int * middle = std::partition(begin, end,
26                                     std::bind2nd(std::less<int>(), *end));

28         std::swap(*end, *middle); // pivot to middle
29         cilk_spawn sample_qsort(begin, middle);
30         sample_qsort(++middle, ++end); // Exclude pivot
31         cilk_sync;
32     }
33 }

```

前述のサンプルコードは簡易版のキーワードを使用して書き直すことができます。ヘッダーファイル `<cilk/cilk.h>` をインクルードします。このヘッダーファイルには、アンダースコア(_)なしの小文字で記述できるマクロが定義されています。次のコードは、簡易版の `cilk_spawn` と `cilk_sync` を利用した記述例です。以降の例では、この名前変換を用いて説明します。

```

19 #include <cilk/cilk.h>

21 void sample_qsort(int * begin, int * end)
22 {
23     if (begin != end) {
24         --end; // Exclude last element (pivot)
25         int * middle = std::partition(begin, end,
26                                     std::bind2nd(std::less<int>(), *end));

28         std::swap(*end, *middle); // pivot to middle
29         cilk_spawn sample_qsort(begin, middle);
30         sample_qsort(++middle, ++end); // Exclude pivot
31         cilk_sync;
32     }
33 }

```

前述の 2 つの例では、29 行目のスポンで `sample_qsort` を非同期に再帰呼び出ししています。従って、29行目が完了する前に 30 行目の `sample_qsort` が再度呼び出されるかもしれません。31 行目の `cilk_sync` ステートメントは、この関数内のすべての `cilk_spawn` で呼び出された処理が完了するまで、ここで待機することを指示します。

各関数の終端には暗黙の `cilk_sync` が隠されており、関数内でスポンされたすべてのタスクがリターンするのを待ちます。31行目の `cilk_sync` は冗長的ですが、明確にするためこの位置に記述されます。

このような変更は再帰呼び出しを行うアルゴリズムを並列化する際の、一般的な「分割統治」の実装例です。それぞれのレベルにおける再帰には、2つの並列連鎖があります。親の連鎖(29行)は現在の関数を継続して実行し、子の連鎖は他の再帰呼び出しを実行します。この再帰は大量の並列関係をもたらします。

実行とテスト

ここまでの変更で、インテル® Cilk™ Plus バージョンの `qsort` プログラムをビルドして実行できるようになりました。次の手順でプログラムをビルドし実行してみます：

Linux * OS: `icc qsort.coo -o qsort`

Windows* コマンドライン: `icl qsort.cpp`

Windows Visual Studio*: Release モードでビルドします。

コマンドラインから `qsort` を実行：

```

>qsort
Sorting 10000000 integers
Sort succeeded.

```

マルチコア・システムでスピードアップを確認

インテル® Cilk™ プログラムは、実行時オペレーティング・システムに利用可能なコア数を問い合わせ、デフォルトで利用可能なすべてのコアを使用します。CILK_NWORKERS 環境変数を利用してワーカーの数を制御することができます。

qsort をシングルコアとデュアルコアで実行してみます。システムが2つ以上のコアを搭載する場合、2回目の実行時間は最初の半分になることが想定されます。

Linux* OS:

```
>CILK_NWORKERS=1 qsort
Sorting 10000000 integers
Sort succeeded.
>CILK_NWORKERS=2 qsort
Sorting 10000000 integers
Sort succeeded.
```

Windows* コマンドライン:

```
>set CILK_NWORKERS=1
>qsort
Sorting 10000000 integers
Sort succeeded.
>set CILK_NWORKERS=2
>qsort
Sorting 10000000 integers
Sort succeeded.
```

インテル® Cilk™ プログラムのビルド、実行およびデバッグ

この節では、インテル® Cilk™ プログラムのワーカー数設定、シリアル実行およびデバッグの方針について説明します。

ワーカー数の設定

ワーカースレッドの数は、デフォルトでホストシステムのプロセッサ・コア数が設定されます。ほとんどの場合デフォルト設定で適切に動作しますが、環境変数を利用してワーカーの数を増減できます。

テスト実行や他のプログラムのリソースを確保するため、利用可能なプロセッサ・コア数より少ないワーカーを利用することもあります。また、プロセッサ・コア数よりも多いワーカーでオーバーサブスクライブな状態を再現することもあるでしょう。さらにこの機能は、ワーカーがロックで待機している場合や、アプリケーションをシングルコア・システム上でテストする際に有効です。

環境変数

環境変数 `CILK_NWORKERS` にワーカースレッドの数を指定します。

Windows* OS: `set CILK_NWORKERS=4`

Linux* OS: `export CILK_NWORKERS=4`

プログラムから制御

インテル® Cilk™ Plus プログラムから `__cilkrts_set_param` API を利用してワーカー数を制御できます。`__cilkrts_set_param` はプログラムの中で最初のタスクスポン（`cilk_swapen` や `cilk_for`）が実行される前に呼び出さなければいけません。

`__cilkrts_set_param("nworkers", "N")` の最初の引数は、環境変数 `CILK_NWORKERS` を変更することを、2 番目の引数（ここではN）には10進、16進（0x）もしくは8進数（0）でワーカー数を指定します。この API は環境変数 `CILK_NWORKERS` を上書きします。この API を利用するには、`<cilk/cilk_api.h>` をインクルードする必要があります。

API の利用例:

```
if (0!=__cilkrts_set_param("nworkers", "4")) // API は正しく設定できると 0 を返す
{
    printf("Failed to set worker count¥n");
    return 1;
}
```

シリアル実行

インテル® Cilk™ Plus で並列化したプログラムを、並列化する前の C/C++ プログラムと同じセマンティックで実行することをシリアル化と呼びます。シリアル化はインテル® Cilk™ Plus プログラムをデバッグする際に役立ちます。

シリアル化する方法

ヘッダーファイル `cilk_stub.h` は、インテル® Cilk™ Plus で使用するキーワードを再定義するマクロを含んでおり、再定義されたライブラリー呼び出しはシリアルに実行されます。シリアル実行を行う前に、`cilk/cilk_stub.h` をインクルードしてビルドを行います。

Linux* OS:

インテル® C++ コンパイラー Linux 版のコマンドライン・オプションには、シリアル実行を制御するため 2 つのオプションが用意されています。

```
-cilk-serialize  
-include cilk/cilk_stub.h
```

インテル® C++ コンパイラーのサンプルコードに含まれる `linear-recurrence.cpp` (インテル® Cilk™ Plus キーワードを含んでいます) をシリアル化してみます。

```
icc -O2 -cilk-serialize -o reducer_serial linear-recurrence.cpp
```

Windows* OS:

インテル® C++ コンパイラー Windows 版のコマンドライン・オプションには、シリアル実行を制御するため 2 つのオプションが用意されています。

```
/Qcilk-serialize  
/FI cilk/cilk_stub.h
```

インテル® C++ コンパイラーのサンプルコードに含まれる `linear-recurrence.cpp` (インテル® Cilk™ Plus キーワードを含んでいます) をシリアル化してみます。

```
icl /O2 /Qcilk-serialize linear-recurrence.cpp
```

注) 通常 `cilk-serialize` オプションを利用する場合、`cilk_stub.h` をソースファイルでインクルードする必要はありません。コンパイラーが自動的に展開します。

デバッグの方針

並列プログラムのデバッグは、シリアルプログラムに比べると難しくなる傾向があります。インテル® Cilk™ Plus では、並列プログラムのデバッグを可能な限り簡単にしよう設計されています。

次に示すガイドラインは並列プログラムにおける問題を軽減します。

- 既存の C/C++ を並列化するならば、事前にシリアルプログラムのデバッグとテストを完了します。
- インテル® Cilk™ Plus を実装したら、最初にシリアル化によってデバッグとテストを行います。シリアルプログラムとインテル® Cilk™ Plus のシリアル化によるプログラムは、シングルスレッドの C/C++ プログラムであるため、既存のデバッグツールや知識が活用できます。
- インテル® Cilk™ プログラム向けに拡張されたLinux* の gdb や Microsoft Visual Studio* などの標準的なデバッガーを利用できますが、結果を解釈するのは困難です。
- 並列化後のプログラムが 1 つのワーカーで正しく動作し、複数のワーカーでおかしな振る舞いをするならば、おそらくデータ競合が発生しています。そのような場合は、次の手法を考慮してください；
 - ソースコードを変更し競合を排除する
 - レデューサーを利用する
 - 排他ロック(インテル® スレッディング・ビルディング・ブロックには mutex ロックが用意されています)、他のロックもしくはアトミック操作を行う

最適化を無効にしたプログラムをデバッグするのは容易です。最適化を無効にすると、インライン展開が行われず、より正確な呼び出しスタックを参照できます。さらに、最適化を無効にするとコンパイラーは命令の再配置(並べ替えや)レジスター割り当ての最適化を行いません。

インテル® Cilk™ Plus 言語機能

この節ではインテル® Cilk™ Plus のキーワード、プリAGMA、事前定義マクロ、環境変数およびコンパイル時のオプションについて説明します。

インテル® Cilk™ Plus 言語機能のまとめ

インテル® Cilk™ Plus にはキーワード、コマンドライン・オプション、環境変数やプリAGMAなど一連の関連項目があります。つぎにこれらの項目をまとめています。

項目	説明
キーワード:	
<code>_Cilk_spawn</code>	関数呼び出しが、呼び出し元と並列動作することをランタイムシステムに指示します。
<code>_Cilk_sync</code>	現在の関数がスポンされた子タスクが完了するまで現在の位置で待機することを指示します。
<code>_Cilk_for</code>	指定した C/C++ の通常ループを置き換えて、ループの各反復を並列処理することを指示します。 このステートメントは、1 つのループをチャンクに分割します。チャンクは 1 つ以上のループ反復から構成されます。各チャンク自身はシリアル実行され、ループが実行される間チャンクはスポンされることで並列に動作します。
プリAGMA:	
<code>cilk grainsize</code>	<code>Cilk_for</code> ループで並列処理されるの 1 回あたりの反復のサイズ(粒度サイズ)を指定します。
事前定義マクロ:	
<code>__cilk</code>	バージョン番号を返します。このリリースでは 200 に設定されています。
環境変数:	
<code>CILK_NWORKERS</code>	ワーカースレッドの数を指定します。
コンパイラーオプション:	
<code>/Qcilk-serialize!</code> (Windows) <code>-cilk-serialize</code> (Linux)	インテル® Cilk™ Plus プログラムのシリアル化を指定します。
<code>/Qintel-extensions[-]</code> (Windows) <code>-[no]intel-extensions</code> (Linux)	インテル® Cilk™ Plus を含むインテル言語拡張を、有効もしくは無効にします。デフォルトではインテル言語拡張は有効です。
ヘッダーファイル (include/cilk に配置):	

cilk.h	簡易版の名称 (cilk_spawn、cilk_sync、cilk_for) を利用するためのマクロを定義します。
cilk_api.h	ランタイム関数とクラスを定義します。
cilk_stub.h	インテル® Cilk™ Plus アプリケーションをシリアル化する場合に使用します。
reducer.h	共通レデューサー定義を含んでいます。
reducer_list.h reducer_max.h reducer_min.h reducer_opadd.h reducer_opand.h reducer_opor.h reducer_opxor.h reducer_ostream.h reducer_string.h	レデューサー・ライブラリーを提供します。レデューサー・ライブラリーの節を参照してください。

キーワード

この節ではインテル® Cilk™ Plus でサポートされる 3 つのキーワード、`_Cilk_spawn`、`_Cilk_sync`、`_Cilk_for` について説明します。

ヘッダーファイル `<cilk/cilk.h>` は簡易版のキーワード (`cilk_spawn`、`cilk_sync`、`cilk_for`) を利用するためのマクロを含んでいます。この節では `cilk.h` に含まれる簡易版のキーワードを利用します。

cilk_spawn

`cilk_spawn` キーワードは、指示されたステートメントの関数呼び出しが、呼び出し元と並列動作することをランタイムシステムに通知します。`cilk_spawn` ステートメントは次のいずれかのフォーマットです。

```
type var = cilk_spawn func args) ; // func () は戻り値を返す
var = cilk_spawn func(args) ; // func () は戻り値を返す
cilk_spawn func(args) ; // func () は void を返す
```

`func` は現在のストランドと並列に動作する関数の名前です。ストランドとは、並列動作しないシリアル動作する命令の並びです。実行中の関数から `cilk_spawn` によって指定される関数の一部を並列実行することができます。

`var` は関数によって返される型の変数です。これは関数からの結果を受け取るので、レシーバーと呼ばれます。レシーバーは `void` 型関数には指定できません。

`args` はスポンされる関数への引数です。これらの引数はタスクがスポンされる前に評価されます。参照渡しやアドレス渡しの引数は十分に考慮してください。なぜなら、このような引数は次に `cilk_sync` が実行されるか、スポンされた関数が終了するまで有効であるため、他のタスクによって破壊された内容を参照する可能性があります。これはデータ競合の典型的な例です。

スポンされた関数は、その関数をスポンした関数の「子」と呼ばれます。逆に、`cilk_spawn` ステートメントを実行した関数は、スポンされた関数の「親」と呼ばれます。

関数はどのような数式表現によってもスポンできます。次の例のように、関数へのポインターやメンバー関数のポインターを利用することもできます：

```
var = cilk_spawn (object.*pointer) args);
```

しかし、次のように他の関数をスポンしその戻り値を関数に渡すようなことはできません；

```
g(cilk_spawn f());          // 許されません
```

関数 `f()` と `g()` の両方の呼び出しをスポンするには次のようにします。ここでは C++ のラムダ関数を利用しています：

```
cilk_spawn [&]{ g(f()); }();
```

上記の表現は、次の方法とは異なることに注意してください：

```
cilk_spawn g(f());
```

後者では `f()` は親関数で実行され、その戻り値を引数として `g()` がスポンされます。前者の関数では、`f()` と `g()` は共にスポンされた子で実行されます。

cilk_sync

`cilk_sync` ステートメントは、現在の関数がスポンされた子タスクが完了するまで現在の位置で待機することを指示します。すべての子タスクが終了すると、現在の関数は処理を続行できます。

シンタックスは以下のとおりです：

```
cilk_sync;
```

`cilk_sync` では現在の関数がスポンした子タスクのみを同期します。他の関数からスポンされた子タスクには影響しません。

すべての関数や `cilk_spawn` を含む `try` ブロックの終端には暗黙の `cilk_sync` が隠されています。これは次の理由により定義されます；

- プログラムが利用する資源がプログラムの並列性に伴って増長しない事を確実にします。
- データ競合のない並列プログラムが、シリアルプログラムと同じ振る舞いとなることを確実にします。呼び出された関数内部で他の関数をスポンするかどうかにかかわらず、通常の間数呼び出しと同じように振る舞います。
- スランドの実行が現在の関数に影響を及ぼさず、資源の解放に失敗しないようにします。
- 呼び出された関数がリターンするときにはすべての操作を完了します。

cilk_for

cilk_for ステートメントは、指定した C/C++ の通常ループを置き換えて、ループの各反復を並列処理することを指示します。

一般的な cilk_for のシンタックスは次のようになります：

```
cilk_for (初期化; 条件式; インクリメント式)
    ループ本体
```

次の条件が適用されます：

- 初期化パートではループ制御変数を 1 つ定義し、初期化しなければいけません。コンストラクターの構文形式は重要ではありません。変数型がデフォルト・コンストラクターならば、明確な初期値は必要ありません。
- 条件式では次のオペレーターの何れかを使用して、制御変数と終了式を比較しなければいけません：
`<=` `!=` `>=` `><`
- 終了式と制御変数は比較オペレーターの左右どちらに置くこともできますが、制御変数は終了式中に記述することはできません。また、終了式の値はループ反復間で変更してはいけません。
- インクリメント式では次のオペレーターの何れかを使用して、制御変数の値を加算もしくは減算します：
`+=`
`-=`
`++` (プリフィックスもしくはサフィックス)
`--` (プリフィックスもしくはサフィックス)

ループ終了式と同様に、インクリメント式における加算値(もしくは減算値)はループ間で変わってはいけません。

実行時インテル® Cilk™ Plus のランタイム・ライブラリーは、cilk_for で指示されたループをループ反復ごとに分割統治に元づいて再帰処理に変換します。

cilk_for ループの利用例を示します：

```
cilk_for (int i = begin; i < end; i += 2)
    f(i);
```

```
cilk_for (T::iterator i(vec.begin()); i != vec.end(); ++i)
    g(i);
```

C 言語の場合のみ制御変数の宣言をループ前に記述できます：

```
int i;
cilk_for (i = begin; i < end; i += 2)
    f(i);
```

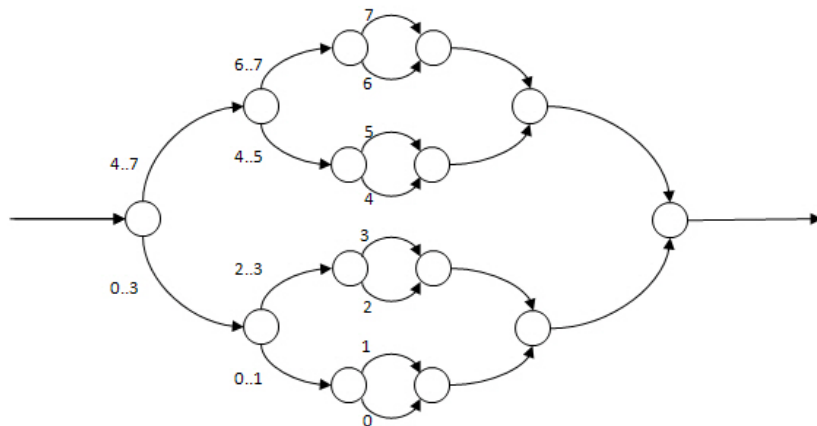
シリアル化されたインテル® Cilk™ Plus プログラムは、C/C++ プログラムと同じ振る舞いをします。シリアル化は、cilk_for を通常の for に置き換えるのと同じ結果となります。cilk_for を構成するループは有効な C/C++ ループでなければいけません。cilk_for ループにはいくつかの制限があるため、C/C++ と比較しました。

ループ本体が並列実行されるため、制御変数をループ内で変更してはいけません。さらにデータ競合を引き起こすようなグローバル変数の変更は避けなければいけません。データ競合を防ぐにはレデューサーを使用できます。

cilk_for におけるシリアルおよび並列実行の構造

cilk_for を利用すると、各ループ反復をスポンするのと同じではありません。実際、インテル® コンパイラーは、ループ本体を分割統治法による再帰呼び出し可能な関数に変換します。この手法は非常に優れたパフォーマンスを実現できます。2 つの方針において、明確な DAG (Directed Acyclic Graph) の違いが見られます。

では最初に反復 N が 8 で、粒度が 1 の場合の、cilk_for ループにおける DAG を考えてみましょう。次の図中の数値は、ストランドと呼ばれるシリアル実行を行う命令のシーケンスを示します。これらの数値は、どのループ反復が、どのストランドで実行されるかを示しています。

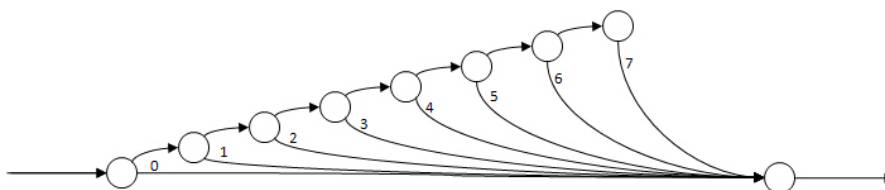


分割では、全体を半分にし一方を子タスクが、残りを親タスクが継続して処理します。両方のループのオーバーヘッドと新たなタスクをスポンするオーバーヘッドは、ループ本体のコストと共に均等に分割されることが重要です。

各ループ反復が同じ時間 T で実行されるなら、プログラムの始まりから終わりまでの最も時間のかかるパスは、8 回のループの場合 $\log^2(N) * T$ もしくは $3 * T$ です。ランタイムは、ループの反復数やワーカーの数にかかわらず、効率よくバランスをとります。

シリアルループ内でスポンされた場合のシリアルおよび並列実行の構造

ここではシリアルループの各反復でスポンを行った場合の DAG を考えてみます。この場合、各子タスクは 1 回のループ反復のみを処理するためワーカーの負荷は均等でなく、同期固有のスケジュールのオーバーヘッドが現れます。小さなループや、ループ本体が制御やスポンのオーバーヘッドよりも時間を要する場合、性能差はほとんどないでしょう。しかし、ループが時間を要さない多くの反復で構成されている場合、オーバーヘッドのコストはいかなる並列化の利点をも打ち消すでしょう。



cilk_for の本体

cilk_for のループ本体は、cilk_for と cilk_sync ステートメント間の範囲を制限する特別な領域が定義されます。cilk_for 構文内の cilk_sync ステートメントは、同じループ反復内でスポンされた子タスクの終了を待ちます。前後の他のループ反復や周辺の他の関数の子タスクとは同期しません。さらに、各ループの終端(ブロック化された可変デストラクターが含まれる)には、暗黙の cilk_sync が隠されています。その結果、関数が cilk_for のループに入る時に終了していない子があれば cilk_for ループを出るときに同じ子が実行されるでしょう。cilk_for ループ内でスポンされた子は、ループが終了する前に同期されることが保証されます。言い換えると、ループに入る前に存在する子は、どれもループが実行される間は同期されません。この cilk_for ループの性質は開発者のプログラムの利点とすることができます(例外ハンドリングを参照)。

cilk_for ループ内で例外がスローされ同じ反復内で例外が処理されなかった場合、いくつかのループ反復は実行されないかもしれません。シリアル実行における例外と異なり、どの反復が実行され、どれが実行されないかは完全には予測できません。例外をスローする場合を除き、ループ反復処理中は中断できません。

Windows* OS: Microsoft が例外処理を構造化しているため制限があります(特に、/EHa コンパイラー・オプションと C/C++ の __try、__except、__finally および __leave 拡張)。例外ハンドリングの Windows* における構造化例外ハンドルを参照してください。

cilk_for 型の要件

cilk_for ループの制御変数として C++ のカスタムデータ型を使用する場合は注意してください。ランタイムシステムがループの範囲を分割することができるよう、各カスタムデータ型をランタイムに通知するメソッドを用意しなければいけません。整数型や STL のランダムアクセス・イテレーターにはあらかじめ異なる型が定義されているので、何も加える必要はありません。

制御変数が variable_type 型で宣言され、ループ終了式が termination_type 型で定義されているとすると、次のようになります;

```
extern termination_type end;
extern int incr;
cilk_for (variable_type var; var != end; var += incr) ;
```

ループが何回実行されるかをコンパイラーに通知するため、いくつかの関数を用意しなければいけません。これらの関数により、コンパイラーは variable_type と termination_type 変数の整数差を計算できます。

```
difference_type operator-(termination_type, variable_type);
difference_type operator-(variable_type, termination_type);
```

次の条件が適用されます:

- 引数型は明確である必要はありませんが、variable_type や termination_type から変換できなければいけません。
- ループがカウントアップできるなら最初のフォームオペレーターが必要です。またループがカウントダウンされるなら 2 番目のオペレーターが必要です。
- 引数はコンスタント参照もしくは値で渡されます。
- プログラムは、ランタイム時にインクリメントがプラスかマイナスによって 1 つもしくは他の関数を呼び出します。
- difference_type の戻り値としてどのような整数値も選択できますが、両方の関数で同じでなければ

いけません。

- `difference_type` の符号の有無は重要ではありません。

以下のように定義することで、システムに制御変数を加算する方法を通知できます：

```
variable_type::operator+=(difference_type);
```

ループ中で `+=` や `++` に変わり `-=` や `-` を記述する場合、`operator++` に変えて `operator-=` を定義します。

これらのオペレーター関数は一般的な演算でなければいけません。コンパイラーは 1 を 2 度加えることを、2 を 1 度加えると想定します。つまり、

`X - Y == 10`

は

`Y + 10 == 10`

と想定されます。

cilk_for の制限

ループの並列化に際し分割統治による技術を利用するため、ランタイムシステムはループ反復の総数を事前に計算しなければならず、ループ制御変数の値はループ反復ごとに求められなければいけません。この計算を可能にするため、たとえユーザー定義型であっても、制御変数は整数として加算、減算および比較が出来なくてはなりません。標準テンプレート・ライブラリーからの整数、ポインターおよびランダムアクセス・イテレーターはすべて整数として振る舞うためこの要件を満たします。

さらに `cilk_for` ループには次の制限があります（標準の C/C++ ループではこの制限はありません）。コンパイラーは以下の違反にエラーもしくは警告を發します。

- ループ制御変数は 1 つでなければならず、初期化では必ず値を設定しなければいけません。次の形式はサポートされません。

```
cilk_for (unsigned int i, j = 42; j < 1; i++, j++)
```

- ループ制御変数はループ本体で変更してはいけません。次の形式はサポートされません。

```
cilk_for (unsigned int i = 1; i < 16; ++i) i = f();
```

- 終了とインクリメント値はループが開始されるときに 1 度だけ評価されます、各ループ反復で再評価してはいけません。さらに、ループ内でこれらの値を変更すると、反復は正しく動作しなくなります。

```
cilk_for (unsigned int i = 1; i < x; ++i) x = f();
```

- C++ の場合、制御変数はループのヘッダー内で定義されなければいけません。ループ外部では定義できません。次の形式は C ではサポートされますが、C++ ではエラーになります。

```
int i; cilk_for (i = 0; i < 100; i++)
```

- `cilk_for` ループ内に `break` および `return` 文は記述できません。コンパイラーはエラーを生成します。このコンテキスト内の `break` と `return` 文は将来の投機的並列向けに予約されています。

- goto 文は cilk_for ループ内でのみ使用できます。分岐先も同じループ内になければいけません。cilk_for ループから外へ分岐したり、外部からループ内への goto 文がある場合、コンパイラーはエラーメッセージを生成します。

- cilk_for ループはラップアラウンドしません。例えば、C/C++では次のように記述できます。

```
(unsigned int i = 0; i != 1; i += 3);
```

C/C++ ループではこれは動作しますが、ループは 2,863,311,531 回実行されることになります。このようなループを Cilk_for ループに変換した場合、予測できない結果をもたらします。

- cilk_for ループは無限ループであってははいけません。

```
cilk_for (unsigned int i = 0; i != i; i += 0);
```

cilk_for の粒度

cilk_for ステートメントはループを 1 つか幾つかのループ反復で構成されるチャンクに分割します。各チャンク自身はシリアルに実行され、ループ実行中チャンクとしてスプーンされます。各チャンクの最大反復数は粒度 (Grain Size) となります。

ループが多くの反復で構成される場合、比較的大きな粒度でオーバーヘッドを軽減できます。そしてループ反復が少ない場合、小さな粒度でプログラムの並列性を高め、プロセッサ・コア数の増加に従ってパフォーマンスが向上します。

粒度の設定

cilk grainsize プラグマを利用して、プログラム中で cilk_for ループの粒度を指定できます。

```
#pragma cilk grainsize = expression
```

次の例のように利用します：

```
#pragma cilk grainsize = 1
cilk_for (int i=0; i<IMAX; ++i) { . . . }
```

粒度を指定しない場合、ランタイムシステムが自動的に粒度を計算します。デフォルト値は次のプラグマが設定されたように振る舞います：

```
#pragma cilk grainsize = min(512, N / (8*p))
```

ここで N はループの反復数で、p はプログラムの実行中に生成されるワーカーの数です。この計算式では 8 から 512 までの並列性に対応します。ループ反復が少ない(8 * ワーカー以下)場合、粒度は 1 に設定され、各反復は並列に動作します。ループ反復が (4096 * p) よりも大きな場合は、粒度は 512 となります。

粒度に 0 (ゼロ) を設定するとデフォルト式が使用されます。0 以下の粒度を指定すると、実行結果は不定です。

プラグマで指定される式はランタイム実行時に評価されるため、次のようにワーカー数を元に粒度を計算することができます：

```
#pragma cilk grainsize = n/(4*__cilkrts_get_nworkers())
```

実行時のループ分割

実行されるチャンクは、ループ反復数 N を粒度 K で割った数になります。

インテル® コンパイラーはループを実行するため分割統治による再帰を行います。制御構造は次のようになります：

```
void run_loop(first, last)
{
  if (last - first) < grainsize)
  {
    for (int i=first; i<last ++i) LOOP_BODY;
  }
  else
  {
    int mid = (last-first)/2;
    cilk_spawn run_loop(first, mid);
    run_loop(mid, last);
  }
}
```

言い換えると、ループ反復の大きさが粒度以下になるまで、ループは繰り返し半分に分割されます。実際の反復数が実行されるので、しばしばチャンクが粒度以下になることがあります。

ループ反復 16 の `cilk_for` を考えてみます：

```
cilk_for (int i=0; i<16; ++i) { ... }
```

粒度が 4 の場合、4 つの反復を持つチャンクが 4 回実行されます。しかし粒度が 5 になると、異なる反復数を持つ 4 つのチャンク(それぞれ 5、3、5、3)に分割されます。

アルゴリズムを詳しく見ると、同じ大きさの 16 回のループがあり、2 と 3 の粒度による結果はそれぞれ 2 回の反復からなる 8 つのチャンクと同じであることが判ります。

最適な粒度を求める

多くの場合デフォルトの粒度で適切に実行できます。次に粒度を決定するガイドラインを示します：

- 1 回当たりのループの処理量にばらつきがあり、処理量の多いループ反復が不規則に分配される場合、最も時間のかかるチャンクが他のチャンクの終了後に配置され、スチールするワーカーが無いアイドルワーカーが生じる可能性があります。
- ループあたりの処理量が少ない場合、粒度を大きくすることを考慮するかもしれません。しかし、このような状況ではデフォルト設定が適切です。並列性を妨げるリスクを冒す必要はありません。
- 粒度を変更するならば、ループが高速に実行されることを検証します。

プリプロセッサ・マクロ

`__cilk` マクロはインテル® コンパイラーによって自動的に定義され、インテル® Cilk™ Plus のバージョンが設定されます。このリリースでは 200 であり、言語バージョンが 2.0 であることを示します。

インテル® Cilk™ Plus の実行モデル

この節ではインテル® Cilk™ Plus の並列実行モデルと例外処理について説明します。

重要な概念

インテル® Cilk™ Plus によるプログラミングは、伝統的なシリアルプログラムに別れを告げ、新たな並列プログラミングへの旅立ちであると言えます。開発者がパフォーマンスの高い、スケールするプログラムを記述できるよう重要な概念を説明することが要求されます。

次に重要な概念を示します：

- ストランド： インテル® Cilk™ Plus プログラムの構造は、並列制御接続されたストランドの集まりであると言えます。
- ワーク、スパン、並列性： インテル® Cilk™ Plus プログラムから期待できるパフォーマンスは、ワーク、スパン、そして並列性を分析することです。

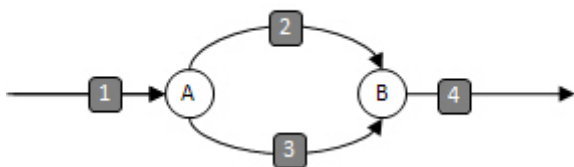
ストランド

伝統的なシリアルプログラムは、コールグラフやクラス階層によって構成され、並列プログラムはシリアル階層に、他のレイヤーを加えて構成されます。インテル® Cilk™ Plus プログラムのパフォーマンスを解析して理解するため、シリアル実行するコード領域と並列実行するコード領域を明確にする必要があります。

ストランドはプログラムのシリアル領域を指す用語です。厳密に言うと、「並列制御の構造を持たない命令のシーケンス」という意味になります。

この定義によるとシリアルプログラムは、プログラム全体が短い命令であるシーケンシャルストランドからなる 1 つの大きなストランドであると言えます。最大のストランドは、始まりと終わりが並列制御構造である、長いストランドの一部分ではないストランドです。

インテル® Cilk™ Plus プログラムは 2 つの並列制御構造 (spawn と sync) から構成されます (cilk_for のような並列ループは、問題を spawn と sync に分解する便利な方法です)。次の図は 4 つのストランド (1,2,3,4)、spawn(A)、そして sync(B) を表しています。この図では、ストランド 2 と 3 のみが並列実行されます。



この図で、ストランドは矢印付きの直線と曲線で示され、丸で囲まれた A と B は並列制御構造です。このストランド図はプログラムのシリアルと並列領域からなる構造を示す DAG (Directed Acyclic Graph) を表します。

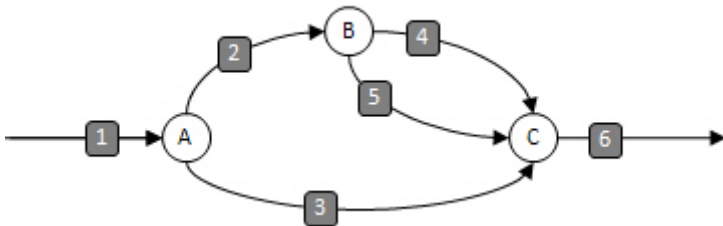
上記の図のインテル® Cilk™ Plus プログラム構造は次のようになります：

```
do_stuff_1(); // ストランド 1 を実行
```

```

_Cilk_spawn func_3(); // ポイント A でストランド 3 をスポーン
do_stuff_2();        // ストランド 2 を実行
_Cilk_sync;          // ポイント B で同期
do_stuff_4();        // ストランド 4 を実行
    
```

インテル® Cilk™ Plus プログラムにおいて、スポーンは 1 つのストランドを入力とし、2 つのストランドを出力します。同期(sync)は、2 つ以上のストランド入力と 1 つのストランド出力を持ちます。次の図は、2 つのスポーン(A と B)と 1 つの同期(C)で構成される DAG を示します。このプログラムの場合、ストランド(2)と(3)の組み合わせと、ストランド(3)、(4)そして(5)の組み合わせが並列実行される可能性を示しています。



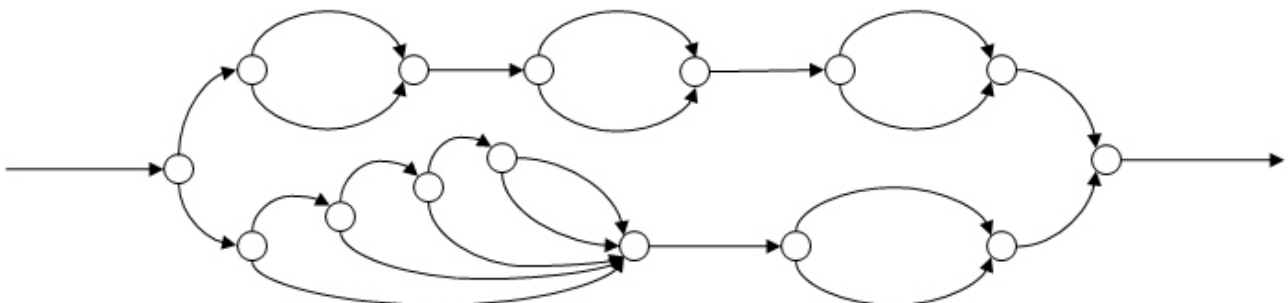
DAG は、インテル® Cilk™ Plus プログラムのシリアルおよび並列実行構造を表しています。同じプログラムでも入力が異なると DAG は変わるかもしれません。例えば、スポーンが条件付きで実行される可能性もあります。

しかし、DAG はプログラムが動作するシステムのプロセッサ数には依存しません。実際、インテル® Cilk™ Plus プログラムがシングルコア(プロセッサ)上で実行される場合も判断できます。以降の節では、プログラムの実行モデルと、作業がどのように利用可能なプロセッサに分割されるか説明します。

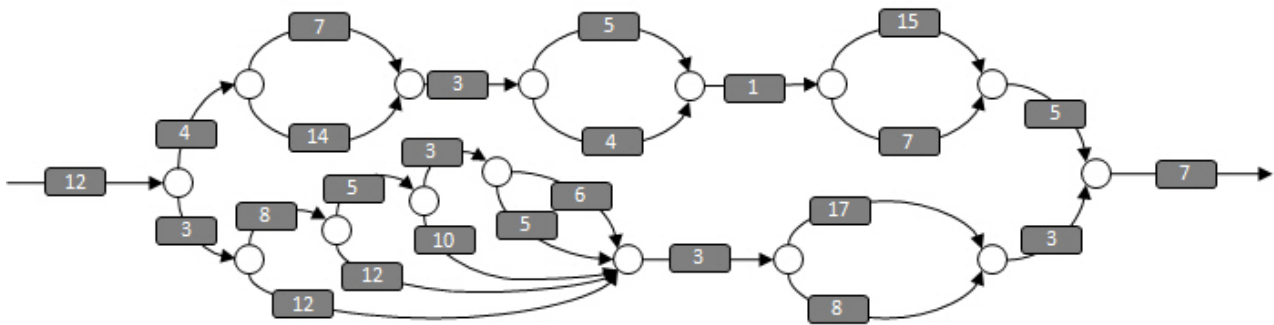
ワーク(work)とスパン(span)

インテル® Cilk™ Plus プログラムのシリアルおよび並列構造を理解するには、並列プログラムのパフォーマンスとスケラビリティを解析することから始めます。

次の図でより複雑なプログラムを考えてみましょう:



この DAG は幾つかのインテル® Cilk™ Plus プログラムの並列構造を表しています。この DAG から成るプログラムを組み立てるため、ストランドにラベル付けして各ストランドの実行に要する時間(ミリ秒)を示します:



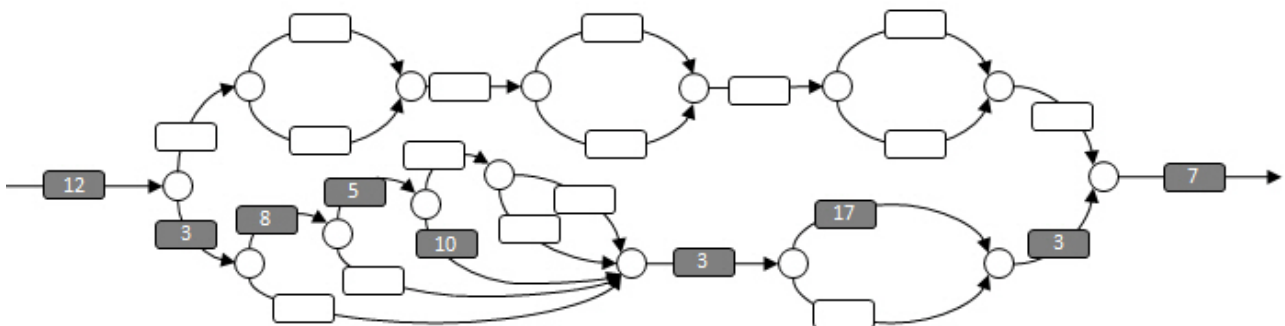
ワーク(work)

プログラムを完了するまでのプロセッサ時間の合計は、図中のすべての値の総和となります。この値をワークとして定義します。

この DAG において、25 個のストランドの作業は 181 ミリ秒であり、プログラムをシングルコア(プロセッサ)で実行すると 181 ミリ秒の実行時間を要することになります。

スパン(span)

スパンはクリティカル・パスと呼ばれることがあり、プログラムの開始位置から終了位置までの間でもっとも時間がかかるパスのことを言います。この DAG においてスパンは 68 ミリ秒であることが、次の図から判ります：



理想的な条件下(例えばスケジュールのオーバーヘッドが無いとか)で、そして無制限のプロセッサ数が利用可能であるとすると、このプログラムは 68 ミリ秒で実行することができます。

インテル® Cilk™ Plus プログラムをどのように高速化し、マルチコア・プロセッサにスケールするか予測する上で、ワークとスパンを利用できます。次の表記が有効となるでしょう：

$T(P)$ は P 個のプロセッサ・コア上のプログラムの実行時間です。

$T(1)$ はワークです。

$T(\infty)$ はスパンです。

デュアルコア・プロセッサの場合、プログラムの実行時間は $T(1) / 2$ 以下にはなりません。一般的に、ワークには次の法則が当てはまります：

$$T(P) \geq T(1) / P$$

同様に、 P 個のプロセッサ・コアでの実行時間は、無限大の個数のプロセッサ・コア上での実行時間以下

にはなりません。スパンには次の法則が当てはまります：

$$T(P) \geq T(\infty)$$

高速化と並列性

プログラムがデュアルコア・プロセッサ上で 2 倍速くなったなら、スピードアップは 2 です。P プロセッサ・コア上でのスピードアップは次のようになります：

$$T(1) / T(P)$$

$T(1)$ が $T(P)$ よりも早く増加するならスピードアップはワークと等しく増加する、という法則は興味深い結果です。例えば特定のアルゴリズムでは、増加したプロセッサ・コアを利用するには処理の追加が必要です。そのようなアルゴリズムは、一般的に 1 もしくは 2 つのプロセッサ・コアではシリアル・アルゴリズムより低速です。しかし、3 つ以上のプロセッサ・コアではスピードアップし始めます。この状況は一般的ではありませんが、気に留める価値はあります。

無制限のプロセッサ・コアを利用可能にすることは、最大限のスピードアップを実現できます。並列性とは、無制限のプロセッサ・コアを想定して、スピードアップを仮定することであると定義できます。並列性は次のように定義できます：

$$T(1) / T(\infty)$$

並列性には期待できるスピードアップに上限があります。例えば、並列化したコードが 2.7 の並列性を持っているなら、そのコードは 2 もしくは 3 つのプロセッサ・コアで期待通りのスピードアップを達成できます。しかし、4 つのプロセッサ・コアでは 3 コア以上のスピードアップは達成できません。つまり、少ないコア数に調整されたアルゴリズムでは、より多くのコアを持つシステムではスケールしないということです。一般的にスケジューラは常にすべてのプロセッサ・コアをビジー保つことはできないので、並列性が 5 から 10 以下であれば、プロセッサ・コアが利用可能ならニアなスピードアップが期待できます。

ストランドをワーカーに割り当てる

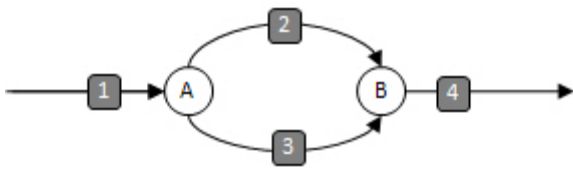
インテル® Cilk™ Plus プログラムにおけるシリアル・並列構造の DAG については前述しました。DAG がプロセッサ・コア数に依存しない事を思い出してください。実行モデルはランタイム・スケジューラがどのようにストランドをワーカーに割り当てるか説明します。

並列性を実装する場合、複数のストランドが同時に実行されるかもしれません。しかし、インテル® Cilk™ Plus プログラムにおいて、同時に実行されるであろうストランドを並列実行する必要はありません。スケジューラは動的にこの決定を行います。

次のプログラムを考えます：

```
do_init_stuff();           // ストランド 1 を実行
  cilk_spawn func3();       // ストランド 3 (子) をスポン
do_more_stuff();          // ストランド 2 を実行 (継続)
  cilk_sync;
do_final_stuff();         // ストランド 4 を実行
```

上記のコードの DAG は次のようになります。



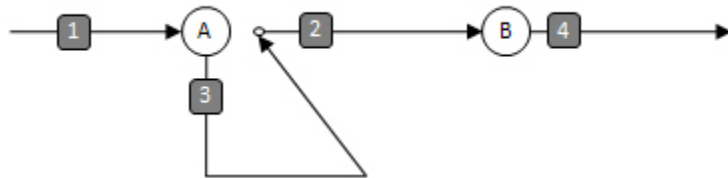
ワーカーはインテル® Cilk™ Plus プログラムを実行するオペレーティング・システムのスレッドであることを思い出してください。1 つ以上のワーカーが利用可能である場合、このプログラムを実行するには次の 2 つの何れかの方法があります：

- プログラム全体を 1 つのワーカーで実行する。
- スケジューラがストランド(2)と(3)を、異なるワーカーで実行する。

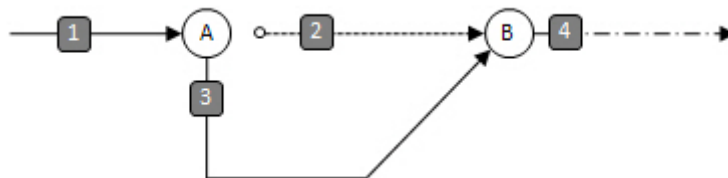
このシリアル実行のセマンティクスを保証するため、スポンされる関数(この例では子、もしくはストランド(3))は、常にスポンされたストランドを実行するワーカーで実行されます。従って、この場合、ストランド(1)と(3)は同じワーカーで実行されることが保証されます。

ストランドを実行可能なワーカーがあれば、ストランド(2)(継続される)は、異なるワーカーで実行されます。これは、スチールと呼ばれ、継続は新しいワーカーにスチールされると言います。

2 つの実行オプションを理解するには次の図が役立ちます。図は 1 つのワーカーで実行される状態を示します：



2 番目のワーカーがスケジュールされると、そのワーカーはストランド(2)の継続実行を開始します。最初のワーカーはポイント B の同期まで実行します。次の図では 2 番目のワーカーは、ストランド(2)として点線で示されます。同期の後、ストランド(4)はいずれかのワーカーで継続して実行されます。現在の実装では、ストランド(4)は、同期に到達した最後のワーカーで実行されます。



実行モデルは、ワーカーとスレッド間のループ反復部とレデューサー部で示される幾つかの実装で成り立ちます。次のことが考えられます：

- `cilk_spawn` された子は常に同じワーカー(システム・スレッド)で実行されます。
- `cilk_spawn` 後の継続は異なるワーカーで実行されます。継続は他のワーカーにスチールされることです。

- `cilk_sync` 後、すべてのワーカーで実行される処理は同期されます。

例外処理

インテル® Cilk™ Plus プログラムは、C++ の例外処理を厳密に再現しようとします。一般的に、例外が保留されている間、並列性は制限されることが要求され、例外処理中プログラムは並列性に依存してはいけません。例外処理の倫理は次のように定義できます：

- スポーンされた子で例外がスローされ、それがキャッチされていないければ、次の同期ポイントで親が再び例外をスローします。
- 親もしくは他の子が例外をスローした場合、シリアル実行でスローされた最初の例外が優先されます。論理遅延例外は捨てられます。同時にスローされた複数の例外をキャッチするメカニズムは今のところありません。

例外をスローする時、例外がスローされる既存の子や兄弟ストランドがアボートすることはありません。それらのストランドは正常に終了します。

`try` ブロックが `cilk_spawn` や `cilk_sync` を含んでいる場合、`try` ブロックの始めと終わりに(デストラクターが起動された後)暗黙の `cilk_sync` が配置されます。同期は `try` ブロック、関数ブロック、そして例外を経由する `cilk_for` 本体を出る前に自動的に実行されます(`cilk_for` 本体内の同期スコープは、同じループ内のスポーンを制限することに注意してください)。キャッチブロックの実行を開始する時点で、関数はアクティブな子を持っていません。これらの暗黙の同期は、プログラムのシリアル処理中の実行と同じようにキャッチされた節が実行されることを確実にします。

暗黙の同期はプログラムの並列性を制限するかもしれません。`try` ブロック前の同期は、次の例のように、`try` ブロックの前にあるスポーンを、早めて同期するかもしれません：

```
void func() {
    cilk_spawn f();
    try { // 暗黙の同期が f() の並列実行を妨げます
        cilk_spawn g();
        h();
    }
    catch (...) {
        // g() や h() の例外処理, しかし f() は対象外
    }
}
```

暗黙の同期が問題となる場合、次の方法で暗黙の同期を防ぐことができます：

- `try` ブロックの本体を他の関数へ移動します。`try` ブロックから `cilk_spawn` を移動することで、ブロックの始まりと終わりで `cilk_sync` が自動生成されることを防ぎます。前述のコード例は、次のように書き換えることができます：

```
void gh()
{
    cilk_spawn g();
    h();
}
```

```
void func() {
    cilk_spawn f();
    try { // cilk_spawn の本体がないため暗黙の同期はない
        gh();
    }
    catch (...) {
        // gh() からの例外を処理、 f()は対象外
    }
}
```

- try/catch ステートメント全体もしくは一度だけ実行される cilk_for ループに入る try ブロック本体だけをブロック化します。cilk_for ループの本体はスポンを含むコンテキストからは分離されるため、同期は周囲の関数には関連しません。このような記述を多用するなら、マクロを利用して次のように書き換えることができます:

```
#define CILK_TRY cilk_for (int _temp = 0; _temp < 1; ++_temp) try

void func() {
    cilk_spawn f();
    CILK_TRY { // try は cilk_for 内部にあるため、f() は同期しない
        cilk_spawn g();
        h();
    }
    catch (...) {
        // g()からの例外を処理、 f()は対象外
    }
}
```

Windows* の構造化された例外処理

Microsoft Windows* の構造化された例外処理には制限があります(コンパイラーの /EHa オプションや、C/C++ 拡張である __try、__except、__finally や __leave を使用する場合)。タスクステールが行われた後と対応する cilk_sync の前で SHE 例外がスローされると、構造化例外処理は(SHE)は失敗します。ランタイムはこの状況を認識して、適切なエラーコードでプログラムを終了します。

レデューサー

この節ではインテル® Cilk™ Plus プログラムにおける競合問題を解決するレデューサーについて説明します。

レデューサーについて

レデューサーは、並列コード内のグローバル変数アクセスに関する問題を解決します。概念的に、レデューサーは並列動作する複数のストランドが安全に利用できる変数です。ランタイムシステムは、各ワーカーがグローバル変数のプライベート・コピーを持ち、ロックを利用することなく競合を排除します。ストランドが同期されるとき、レデューサーは各ワーカーのローカルコピーを元の変数にマージします。ランタイムはオーバーヘッドを最小限にするため、必要な場合にのみコピーを作成します。

レデューサーには次のような特徴があります：

- レデューサーはデータ競合なしでグローバル変数にアクセスすることを可能にします。
- レデューサーはロックを必要としません。従って、ロックによってグローバル変数のアクセスを保護する場合に発生するロックの衝突(それに伴う並列性の損失)を回避できます。
- レデューサーを正しく定義して使用するとシリアル処理のセマンティクスを保つことができます。レデューサーを利用したプログラムの結果は、シリアルバージョンを等しく、プロセッサ・コアの数やワーカーのスケジュールには依存しません。レデューサーは既存のコードを再構築することなく利用できます。
- レデューサーは最小限のオーバーヘッドにより効率良く実装できます。
- レデューサーは、ループなど特異的な制御構造で定義される構文とは異なり、プログラムの制御構造にかかわらず利用できます。

レデューサーは、ランタイムシステムにインターフェースを提供する C++テンプレートで定義されます。

事前定義されたレデューサーに代わって、独自のレデューサーを記述することもできます。詳細は、「新たなレデューサーを記述する」の節を参照してください。

レデューサーを使用してみる

次のコードは、並列動作する総和を求めるコードにレデューサーを利用した例です。compute() 関数を繰り返し呼び出し、total 変数に結果を求めるシリアルプログラムです。

```
#include <iostream>

unsigned int compute(unsigned int i)
{
    return i; // return a value computed from i
}

int main(int argc, char* argv[])
{
    unsigned int n = 1000000;
```

```

unsigned int total = 0;

// Compute the sum of integers 1..n
for(unsigned int i = 1; i <= n; ++i)
{
    total += compute(i);
}

// the sum of the first n integers should be n * (n+1) / 2
unsigned int correct = (n * (n+1)) / 2;

if (total == correct)
    std::cout << "Total (" << total
                << ") is correct" << std::endl;
else
    std::cout << "Total (" << total
                << ") is WRONG, should be "
                << correct << std::endl;
return 0;
}

```

このプログラムをインテル® Cilk™ Plus プログラムに変換するには、for を並列動作する `cilk_for` ループに変更します。そのままでは、並列ループ中でデータ競合が発生します。競合を排除するには、結合 + オペレーター型のトータル・レデューサー - `reducer_opadd` を定義します。次のように変更します。

```

#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>
#include <iostream>

unsigned int compute(unsigned int i)
{
    return i; // return a value computed from i
}

int main(int argc, char* argv[])
{
    unsigned int n = 1000000;
    cilk::reducer_opadd<unsigned int> total;

    // Compute 1..n
    cilk_for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i);
    }

    // the sum of the first N integers should be n * (n+1) / 2
    unsigned int correct = (n * (n+1)) / 2;

    if (total.get_value() == correct)
        std::cout << "Total (" << total.get_value()
                    << ") is correct" << std::endl;
    else
        std::cout << "Total (" << total.get_value()
                    << ") is WRONG, should be "
                    << correct << std::endl;
    return 0;
}

```

```
}

```

レデューサーの実行手順を示します：

1. 該当するレデューサーのヘッダーファイルをインクルードします。
2. `reducer_kind<TYPE>` の形式でレデューサー変数を定義します。
3. 並列実行するため、`for` ループを `cilk_for` ループに変更します。
4. `cilk_for` ループが終了したら、レデューサーの最終値を `get_value()` メソッドを利用して取得します。

注

レデューサーはオブジェクトです。そのため値を直接コピーする事はできません。コピーコンストラクターの代わりに、`memcpy()` を利用して結果をコピーすると予測できない結果をもたらします。

レデューサーが動作する仕組み

レデューサーのメカニズムとセマンティックの説明は、より高度な開発者がレデューサーの利用規則を理解し、独自のレデューサーを記述する背景を理解できます。

もっとも簡単なレデューサー形式は、値、identity 値、リダクション関数などを持つオブジェクトです。

レデューサー・ライブラリーが提供するレデューサーは、レデューサーが安全で一貫した方法で利用できるように追加のインターフェースを提供します。

ここでは、レデューサーが宣言されたときに作成されるオブジェクトは、レデューサーの「左辺」インスタンスとして知られています。

次に簡単な例と、このプログラムがシステム上で実行されるランタイムの背景を説明します。

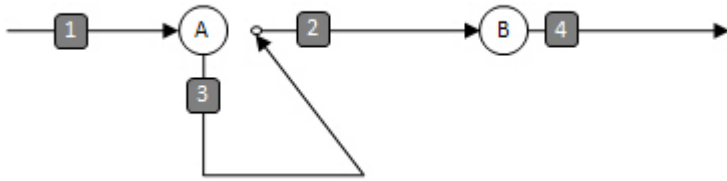
最初に、2 つの `cilk_spawn` による、スチールなしの実行の可能性を考えます。レデューサーの動作はシンプルです：

- スチールが起こらなければ、レデューサーは通常の変数と同じです。
- スチールが起こると、継続は identity 値を受け取り、子はスポンに先行するレデューサーを受け取ります。同期ポイントでは継続中の値は、子がレデューサー操作を利用してレデューサーにマージされます。新規のビューは破壊され、オリジナルが更新され生き残ります。

次の図でこの操作を示します：

スチールが無い場合

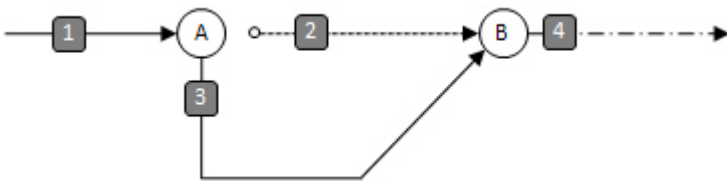
(A) 地点で `cilk_spawn` が実行された後にスチールが無い場合：



この場合、ストランド (1) のレデューサー・オブジェクトは、ストランド (3) で子が参照できます。ストランド (2) は継続し、identity 値と新しいビューを受け取ります。これはスチールされないため、新しいビューは生成されず、レデューサー操作が呼び出されないためです。

スチールが起こる場合

ストランド(2)は、(A) 地点で `cilk_spawn` の継続でありスチールされます。



この場合、ストランド (1) のレデューサー・オブジェクトは、ストランド (3) の子が参照できます。ストランド (2) は継続し、identity 値と新しいビューを受け取ります。同期ポイント (B) で、新しいレデューサーのビューはストランド (3) で参照できる元の値にマージされます。

例: `reducer_opadd` を利用する

この例は、`reducer_opadd<>` を利用して、並列に累積を求めるコードです。identity 値は 0 であり、リダクション機能は右辺値を左辺値に加算します：

```

1 #include <cilk/cilk.h>
2 #include <cilk/reducer_opadd.h>
3
4     cilk::reducer_opadd<int> sum;
5
6     void addsum()
7     {
8         sum += 1;
9     }
10
11     int main()
12     {
13         sum += 1;
14         cilk_spawn addsum();
15         sum += 1; // the value of sum here depends on whether a steal occurred
16         cilk_sync;
17         return sum.get_value();
18     }

```

始めにこのコードが、シングルコア・プロセッサ上でシリアル実行されるのを考えてみます(スチールはもちろんありません)。この場合 `sum` のプライベートビューは生成されません。すべての操作はもっとも左のインスタンスに実行されます。新しいビューは生成されないため、リダクション操作は呼び出されません。`sum` の値は単純に、0 から最後の値である 3 までインクリメントされます。この場合スチールが行われなため、

`cilk_sync` ステートメントは NOP (No operation)として扱われます。

次にスチールが起こる場合を考えてみます。15 行目で `sum` がアクセスされる時、新しいビューの `identity` 値 (0) が生成され、15 行目が実行された後に `sum` の値は 1 になります。親は新しいビュー (`identity`) を取得し、子はスポンされた時にアクティブであったビューを取得します。これは結合できるけれども置き換えできないレデューサーの結果を、レデューサーが決定論的に維持することを許します。子 (`addsum`) は左辺のインスタンスを操作し、8 行目で `sum` は 1 から 2 へ増加します。

`cilk_sync` ステートメントに到達した時に、結合するストランドの `sum` が異なるビューを持っていると、リダクション操作によってマージされます。この場合、リダクション操作は加算であり、親の (1 の値) 新しいビューは、左辺のインスタンスで受け取った 3 の値を受け取る子 (2 の値) のビューへ加算されます。

怠慢なセマンティクス

レデューサーのプライベートビューを持つ各ストランドについて考えます。パフォーマンスの視点から、これらのビューは、次の 2 つの条件が生じた場合に lazily (怠慢な) に生成されます。

- 第 1 に、新しいビューはスチール後に生成される。
- 第 2 に、新しいストランドでレデューサーが最初にアクセスされたときに、新しいビューが生成される。

新しいビューが生成されているなら、`cilk_sync` で事前のビューにマージされます。ビューが生成されていない場合は、リダクションは必要ありません (論理的に `identity` は生成されマージされますが、NOP として扱われます)。

安全な操作

`identity` とリダクション操作だけで、レデューサーを定義することは可能ですが、レデューサーの操作を理解することを制限するため、オペレーターの負荷を利用してより安全で便利な機能を提供します。

例えば、`reducer_opadd` は `+=`、`-=`、`*`、`++`、`--`、`+`、そして `-` オペレーターを定義します。乗算 (`*`) や除算 (`/`) は決定論的で一貫したセマンティクスを提供しないため、`reducer_opadd` では定義されません。

安全性とパフォーマンスに関する考察

一般的にレデューサーは、並列実行間で繰り返し使用される結果を提供する際にロックを必要としません。結果はシリアル実行と同じです。

プログラムを開発する際には、次の安全性とパフォーマンスに関する考察に留意してください。

安全性

厳密な決定論的結果を得るため、レデューサーによって操作 (変更およびマージ) される値は結合されなければいけません。

レデューサー・ライブラリーに定義されているレデューサーは、すべて結合可能です。一般的に、レデューサーにアクセスする際にこれらのオペレーターのみを利用するならば、決定論的なシリアルなセマンティクスを期待できます。結合できない独自のレデューサーを記述した場合や、安全でないオペレーターでレデューサーの値をアクセスもしくは更新するようなケースなど、結合できない操作のレデューサーを使用することはできます。

決定論

レデューサーが浮動小数点型で示される場合、操作は厳密には結合できません。特に値の指数が異なる場合、

操作は異なった結果を生成します。これは、ストランドが実行される順番で結果が異なることに繋がります。いくつかのプログラムにおいて、この違いは許容できますが、プログラムの実行を繰り返す際に同じ結果とならないことがあることに留意してください。

パフォーマンス

レデューサーを適切に利用すると、ランタイム時のパフォーマンス低下はわずかもしくは皆無です。しかし、以下に示す状況では大きなオーバーヘッドがあります。オーバーヘッドはスチールが起こった回数に比例します。

大量のレデューサーを作成する場合（例えば、配列やレデューサーのベクター）、プログラム中のレデューサー数に比例してスチールとレデュースでオーバーヘッドが生じることに注意してください。

大量のステートを伴うレデューサーを定義する場合、スチール後にレデューサーが参照されると、identity 値を作成するのはペナルティーを被るかもしれません。

マージ操作に時間を要するなら、スチールが成功した後の同期ごとにマージが発生している可能性があります。

レデューサー・ライブラリー

レデューサー・ライブラリーには、次の表に一覧されるレデューサーが含まれます。

それぞれのレデューサーの説明は、対応するヘッダーファイルに対応するコメント欄に示されます。

以下の表の中央の欄は、レデューサーの identity 要素と update 操作が説明されます。

レデューサー/ ヘッダーファイル	Identity/ Update	説明
reducer_list_append <cilk/reducer_list.h>	空のリスト push_back()	append 操作で使用するリストを作成します。最終的なリストは、ワーカーの数やワーカーのスケジュール順序にかかわらず、同等のシリアルプログラムの結果と同じになります。
reducer_list_prepend <cilk/reducer_list.h>	空のリスト push_front()	prepend 操作で利用するリストを作成します。
reducer_max <cilk/reducer_max.h>	コンストラクターへの 引数 cilk::max_of	一連の値から最大値を見つけます。コンストラクターの引数は、初期の最大値を持ちます。
reducer_max_index <cilk/reducer_max.h>	コンストラクターへの 引数 cilk::max_of	一連の値から最大値と最大値を含む要素のインデックスを見つけます。コンストラクターの引数は、初期の最大値とインデックスを持ちます。
reducer_min <cilk/reducer_min.h>	コンストラクターへの 引数 cilk::min_of	一連の値から最小値を見つけます。コンストラクターの引数は、初期の最小値を持ちます。
reducer_min_index <cilk/reducer_min.h>	コンストラクターへの 引数 cilk::min_of	一連の値から最小値と最小値を含む要素のインデックスを見つけます。コンストラクターの引数は、初期の最小値とインデックスを持ちます。

レデューサー/ ヘッダーファイル	Identity/ Update	説明
reducer_opadd <cilk/reducer_opadd.h>	0 +=, =, -=, ++, --	総和を求めます。
reducer_opand <cilk/reducer_opand.h>	1 / 真 &, &=, =	論理もしくはビット AND を行います。
reducer_opor <cilk/reducer_opor.h>	0 / 偽 , =, =	論理もしくはビット OR を行います。
reducer_opxor <cilk/reducer_opxor.h>	0 / 偽 ^, ^=, =	論理もしくはビット XOR を行います。
reducer_ostream <cilk/reducer_ostream.h>	コンストラクター << への引数	並列に記述できる出力ストリームを提供します。出力ストリームの一貫した並びを確保するため、現在の位置以前の保留されている出力が無いようにレデューサークラスによってバッファされます。これにより、出力は常にシリアルプログラムと同じ順番になることが保証されます。
reducer_basic_string <cilk/reducer_string.h>	空の文字列、もしくは コンストラクター += と append への引 数	Append や += 操作で使用する文字列生成します。な文字列は、文字列のコピーとメモリの断片化を防ぐためサブ文字列のリストとして管理されます。サブ文字列は、get_value() が呼び出された時に 1 つの出力文字列として組み立てられます。
reducer_string <cilk/reducer_string.h>	空の文字列、もしくは コンストラクター += と append への引 数	char 型の reducer_basic_string への簡略版を提供します。
reducer_wstring <cilk/reducer_string.h>	空の文字列、もしくは コンストラクター += と append への引 数	wchar 型の reducer_basic_string への簡易版を提供します。

C 言語でレデューサーを利用する

前述の例の reducer_opadd<> ライブラリーは C++表記であり、C 言語ではそのまま利用できません。インテル® Cilk™ Plus では、C 言語向けのレデューサー表記として、CILK_C_REDUCER<型>(<変数名>、<変数名>、初期値)を定義します。

例えば、C++における reducer_opadd<int> sum (初期値 0)は、CILK_C_REDUCER_OPADD(sum, int 0) と記述します。

定義したレデューサーにシリアル領域でアクセスする場合は、レデューサーのメンバーを利用しますが、インテル® Cilk™ Plus の並列領域でアクセスするには、REDUCER_VIEW(レデューサー名)を利用します。上記

の例を当てはめると、REDUCER_VIEW(sum)となります。

メモリーリークを避けるため、レデューサーを利用する前に登録、利用し終わったら登録解除します。レデューサーの登録には、CILK_C_REGISTER_REDUCER(sum)を、解除にはCILK_C_UNREGISTER_REDUCER(sum)を利用します。

カスタム・レデューサーの作成:

1. #include <cilk¥reducer.h>

2. レデューサーの振る舞いを定義するため、次のような関数を作成します:

```
void x_identity(void* key, void* v);
void x_reduce(void* key, void* l, void* r);
void x_destroy(void* key, void* p); // 哲稔ル楊豎ネ所躑ル嬌嘯ワ(二所躑
```

3. レデューサーインスタンスを生成します:

```
typedef CILK_C_DECLARE_REDUCER(<data type>) x_reducer;
```

4. 次のシンタックスでレデューサーを宣言および初期化します:

```
x_reducer <reducer name> = CILK_C_INIT_REDUCER(<data type>, x_reducer, x_identity,
x_destroy, <initial value>);
```

5. 前述のレデューサーアクセス規則に従ってレデューサーを利用します。

レデューサーの利用例

この節では String と List レデューサーを含む、幾つかのレデューサーの利用例を示します。

文字列(String)レデューサー

reducer_string は 8 ビットの文字列を組み立て、更新操作として += (文字列結合)などに利用されます。

例ではランタイム時に如何にしてレデューサーがシリアル処理のセマンティクスを保つか説明します。シリアルループにおいて、レデューサーは、'A' から 'Z' までの文字を結合して表示します:

```
The result string is: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

cilk_for ループは、ループを 2 つに分割し、さらにその半分を適度な大きさのチャンクになるまで分割を繰り返す際に、分割統治アルゴリズムを使用します。従って、最初のワーカーは "ABCDEF" を 2 番目のワーカーは "GHIJKLM"、さらに 3 番目のワーカーは "NOPQRS" そして 4 番目のワーカーは "TUVWXYZ" の文字列を持っているかもしれません。ランタイムシステムは、最終結果がアルファベット順に文字列が並んでいるように、レデューサーのレデュースメソッドを呼び出します。

文字列連結は結合しやすい(しかし交換できない);操作の順番は重要ではありません。例えば、次の 2 つの例は同等です:

- "ABCDEF" concat ("GHIJKLM" concat ("NOPQRS" concat "TUVWXYZ"))
- ("ABCDEF" concat "GHIJKLM") concat ("NOPQRS" concat "TUVWXYZ")

cilk_for がどのような順番でワーカーチャンクを生成しても、結果は常に同じです。

get_value() 呼び出しは、サブ文字列をレデュース操作して結合し、1 つの出力文字列に構成します。文字列を取り出すのに、なぜ get_value() を利用するのでしょうか？ 値はいつでも取得できますが、そうすべきではありません。結果は予期しない中間値となり、それは無意味です。この例では、結果は、"GHIJKLM " と "NOPQRS" を結合した "GHIJKLMNOPQRS" でなければいけません。

インテル® Cilk™ Plus のレデューサーはシリアルなセマンティクスを提供します； シリアル・セマンティクスは、cilk_for ループの終端などで、ランタイムシステム無がすべてのレデューサー操作を実行した後、並列領域の最後に保障されます。cilk_for ループ内で get_value() を呼び出してはいけません； ループ中で呼び出された結果をレデュースすると、予測できない無意味な値を得ることになります。

前述の整数加算の例とは異なり、レデュース操作は交換できません。次の例では、レデューサー・ライブラリーの reducer_list_append を使用して要素をリストに追加できます。

```
#include <cilk/cilk.h>
#include <cilk/reducer_string.h>
#include <iostream>
int main()
{
    //...
    cilk::reducer_string result;
    cilk_for (std::size_t i = 'A'; i < 'Z'+1; ++i){
        result += (char)i;
    }
    std::cout << "The result string is: "
                << result.get_value() <<std::endl;
    return 0;
}
```

この例や他の例でも示すように、各ループ反復は一度だけレデューサーを更新します。各反復で複数の更新があります。例えば：

```
cilk_for (std::size_t i = 'A'; i < 'Z'+1; ++i){
    result += (char)i;
    result += tolower((char)i);
}
```

は、次の文字列を生成します。

AaBb...Zz

レデューサーを表示する(ユーザー定義型)

reducer_list_append は、更新操作に STL のリスト append メソッドを利用してリストを生成します。identity は空のリストです。次の例は前述の文字列の例と殆ど同じです。reducer_list_append 宣言は、コードで示される型を必要とします。

```
#include <cilk/cilk.h>
#include <cilk/reducer_list.h>
#include <iostream>
int main()
{
    //...
    cilk::reducer_list_append<char> result;
    cilk_for (std::size_t i = 'A'; i < 'Z'+1; ++i){
```

```

    result.push_back ((char)i);
}

std::cout << "String = ";
std::list<char> r;
r = result.get_value() ;
for (std::list<char>::iterator i = r.begin();
     i != r.end(); ++i){
    std::cout << *i;
}
std::cout << std::endl;
}

```

再帰関数のレデューサー

レデューサーは `cilk_for` ループ中の更新操作で制限されません。レデューサーは再帰スポン関数などの制御フローと共に動作できます。次の例は、ツリー構造の要素リストの並びを、レデューサーがどのように生成するかを示しています。最終結果のリストは、コア数や計算のスケジュールにかかわらず常にシリアル実行と同じ並びの要素を含んでいます。

```

#include <cilk/reducer_list.h>
Node *target;
cilk::reducer_list_append<Node *> output_list;
...
// Output the tree with an in-order walk
void walk (Node *x)
{
    if (NULL == x)
        return;
    cilk_spawn walk (x->left);
    output_list.push_back (x->value);
    walk (x->right);
}

```

高度な利用法： 新しいレデューサーの記述法

提供されるレデューサーに満足できないのであればカスタム・レデューサーを記述できます。

提供されるレデューサーは、新しく作るレデューサーのモデルとして参照できますが、幾つかの例は少々複雑です。それらの実装は、インストール先の `include/cilk` にある各ヘッダーファイル `reducer_*.h` を参照できます。

幾つかの例を以下に示します。

レデューサーの構成要素

レデューサーは、論理的に次の 4 つに分割できます。

- “View” クラス - レデューサーのプライベート・データです。コンストラクターとデストラクターは、ランタイムシステムから呼び出しできるように、パブリックでなければいけません。コンストラクターは、レデューサーの `identity` 値に View を初期化しなければいけません。Identity 値については後述します。View クラスは、それらをプライベート・データのメンバーとしてアクセス、もしくはアクセスメソッドを提供するフレンドを持つレデューサークラスです。

- “Monoid” クラス - monoid は以降に定義する数学上の概念です。ここでは、monoid が `cilk::monoid_base<View>` 由来していなければならない、次の形式で、パブリックな静的メンバー名 `reduce` を含んでいなければならない、ということを理解しておけば十分です:

```
static void reduce (View *left, View *right);
```

- ストランドあたりの view を提供するハイパーオブジェクト。次に示すように、プライベートメンバー変数として定義されなければいけません:

```
private: cilk::reducer<Monoid> imp_;
```

- 残りのレデューサーは、データにアクセスおよび変更するルーチンを提供します。 `get_value()` メンバーはレデューサーの値を返します。

レデューサー・ライブラリーで提供されるレデューサーは、View および Monoid クラス向けに class に変えて struct を利用します。C++ において struct と class の違いは、class のデフォルトアクセスはプライベートですが、struct のデフォルトアクセスはパブリックであることを思い出してください。

Identity 値

identity 値は、2 番目の値を生成するどちらかの並びが別の値に結合されます。

- 0 は加算のための identity 値:

$$x = 0 + x = x + 0$$

- 1 は乗算のための identity 値:

$$x = 1 * x = x * 1$$

- 空の文字列は、文字列結合のための identity 値です:

$$\text{"abc"} = \text{""} \text{ concat } \text{"abc"} = \text{"abc"} \text{ concat } \text{""}$$

Monoid

数学的に monoid は、1 組の値(型)から成り立ち、その組の結合操作、そしてそれらの組と操作のための identity 値を含みます。例えば、(integer, +, 0) や (real, *, 1) は monoid です。

レデューサー・ライブラリーにおいて、monoid は T 型として定義され、次の 5 つの関数を提供します。

<code>reduce(T *left, T *right)</code>	<code>*left = *left OP *right</code> を評価します
<code>identity(T *p)</code>	初期化されていない <code>*p</code> に IDENTITY 値を構成します。
<code>destroy(T *p)</code>	P によって示されるオブジェクトでデストラクターを呼び出します。
<code>allocate(size)</code>	メモリーの <code>size</code> バイトへのポインターを返します。
<code>deallocate(p)</code>	P に割り当てられているメモリーを解放します。

これら 5 つの関数は、static か const 型の何れかでなければいけません。Monoid の必要条件を満たすクラスは、通常状態を持ちません。しかし時々、identity オブジェクトを初期化のに利用される状態を持ちます。

`monoid_base` クラスのテンプレートは、new オペレーターを利用して割り当てる T 型値をデフォルトとする

identity 値のために大きな minoid クラスを持つベース・クラスにおいて有効です。monoid_base に由来するクラスは、レデューサー関数として実装され定義される必要があります。

レデューサー関数は、右のインスタンスから左のインスタンスへデータをマージします。ランタイムシステムがレデューサー関数を呼び出した後、右のインスタンスは破壊されます。

決定論的な結果を得るため、レデューサー関数は結合操作を実装しなければいけません。しかしそれは、置き換える必要はありません。正しく実装されると、レデューサー関数はシリアルなセマンティクスを保ちます。これは、シリアル実行するアプリケーションの結果は、複数のワーカーによるアプリケーションが 1 つのワーカーを利用した場合と同じであることを意味します。ランタイムシステムは、ワーカーやプロセッサの数、そしてストランドがどのようにスケジュールされても、レデューサー関数の結果が等しくなることを確実にします。

レデューサーを記述します - “Holder” の例

この例では、どのようにレデューサーを記述するか説明します。この簡単なレデューサーは、更新メソッドを持たず、レデュースメソッドは何も行いません。“Holder” は “スレッド・ローカル・ストレージ” に似ていますが、OS スレッドの一般的な相互作用の節で説明されるような陥りやすい誤りはありません。

ここでの規則は、完全な同期が故意に破壊される場合を除いて、get_value() 関数を呼び出してはいけないということです。

そのようなレデューサーは実用に向いています。ここで各 for ループ反復で利用されるグローバルなテンポラリーバッファがあることを仮定してください。これはシリアルプログラムでは安全ですが、ループを cilk_for に変換し並列実行する場合はそうではありません。

次のプログラムでは、グローバル一時変数 temp を利用して値を入れ替え、配列全体の要素を反転させることを念頭に置いてください。

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

class point
{
public:
    point() : x_(0), y_(0), valid_(false) {};
    void set (int x, int y) {
        x_ = x;
        y_ = y;
        valid_ = true;
    }

    void reset() { valid_ = false; }

    bool is_valid() { return valid_; }
    int x() { if (valid_) return x_; else return -1; }
    int y() { if (valid_) return y_; else return -1; }

private:
    int x_;
    int y_;
    bool valid_;
};

point temp; // temp is used when swapping two elements

int main(int argc, char **argv)
```

```

{
  int i;
  point ary[100];

  for (i = 0; i < 100; ++i)
    ary[i].set(i, i);

  cilk_for (int j = 0; j < 100 / 2; ++j)
  {
    // reverse the array by swapping 0 and 99, 1 and 98, etc.
    temp.set(ary[j].x(), ary[j].y());
    ary[j].set (ary[100-j-1].x(), ary[100-j-1].y());
    ary[100-j-1].set (temp.x(), temp.y());
  }
  // print the results
  for (i = 0; i < 100; ++i)
    std::printf ("%d: (%d, %d)\n", i, ary[i].x(), ary[i].y());

return 0;
}

```

グローバル変数 `temp` には競合がありますが、シリアルプログラムは正しく動作します。この例では、`cilk_for` の内側で `temp` を宣言することにより、簡単に(そして安全に)問題を解決できます。しかし、ストランドにローカルなストレージを提供することで、以下に示す “Holder” パターンを利用することができます。

ここでのソリューションは、“Holder” レデューサーを実装して利用することです。

`point_holder` クラスはレデューサーであり、`view` としてクラスを指します。`monoid` クラスは、何も行わないレデューサー関数を含みますが、どのバージョンを含んでいるかは重要ではありません。`point_holder` メソッドの残りは、レデューサーデータの更新とアクセスを可能にします。

```

#include <cilk/reducer.h>
class point
{
  // Define the point class here, exactly as above
};

// Define the point_holder reducer
class point_holder
{
  struct Monoid: cilk::monoid_base<point>
  {
    // reduce function does nothing
    static void reduce (point *left, point *right) {}
  };

private:
  cilk::reducer<Monoid> imp_;

public:
  point_holder() : imp_() {}

  void set(int x, int y) {
    point &p = imp_.view();
    p.set(x, y);
  }
};

```

```

}

bool is_valid() { return imp_.view().is_valid(); }

int x() { return imp_.view().x(); }

int y() { return imp_.view().y(); }
}; // class point_holder

```

このサンプル中の `point_holder` レデューサーを利用するには、`temp` 宣言を利用するレデューサー型宣言に置き換えてください。

```
point temp; // temp is used when swapping two elements
```

を

```
point_holder temp; // temp is used when swapping two elements
```

で置き換え、残りのプログラムを変更する必要はありません。

`Point_holder` レデューサーがどのように動作するかまとめます：

1. 新しい `point_holder` が生成されると、デフォルト・コンストラクターは新しいインスタンスを生成します。つまり、`temp` はスチール後に参照されます。
2. レデューサーメソッドは何も行いません。`temp` が一時ストレージとして利用される場合、左右のインスタンスを結合(レデュース)する必要はありません。
3. デフォルトのデストラクターは、メモリーを解放するデストラクターを起動します。
4. ループ反復内のローカル `view` で生成される値は、他の反復の値とは結合されないため、`cilk_for` 内のローカル値を取得するため `x()` と `y()` を呼び出すのは有効です。
5. `Holder` のレデューサー関数は何も行わないので、デフォルト・コンストラクターは真の `identity` 値を提供する必要はありません。

オペレーティング・システムに関する考察

この節ではインテル® Cilk™ Plus とオペレーティング・システムのスレッドの関係を説明します。

インテル® Cilk™ Plus プログラムと他のツールを利用する

インテル® Cilk™ Plus プログラムは、標準 C や C++ とは異なるスタック割り当てと呼び出し規則を持つため、実行可能なバイナリーに適用できる他のツール(メモリーチェッカーやコードカバレッジなど)は、インテル® Cilk™ Plus プログラムには適用できないかもしれません。

しかしプログラムを 1 つのワーカーで実行する(CILK_NWORKERS 環境変数に 1 を設定)ことで解決できる場合もあります。動作しない場合は、インテル® Cilk™ Plus プログラムのシリアル化を行ってください。

OS スレッドとの一般的な相互作用

OS スレッドと共に利用する場合、次の点に注意してください。

ワーカースレッドは OS スレッドであること

ランタイムシステムは一連のワーカースレッドを、ネイティブ OS スレッドを利用して配置します。

インテル® Cilk™ Plus プログラムは常に利用可能なプロセッサを 100% 利用するわけではない

インテル® Cilk™ Plus プログラムを実行する時、プログラムが並列動作していなくても、システム上のすべてのプロセッサを利用しているように見えます。それは、Windows* のタスクマネージャーの“パフォーマンス”タブを使用して計測できます。1 つのストランドしか実行されていないのにすべての CPU がビジーに見えます。

実際には、ランタイム・スケジューラーは CPU を他のプログラムに譲ります。他のプログラムが CPU を要求していない時、ワーカーは直ちにワークスチールを行い、実行を再開します。このため CPU は常にビジーに見えます。従ってプログラムは常に CPU を消費しているように見えますが、これはシステムや他のプログラムへ悪影響を及ぼすことはありません。

ネイティブ・スレッド API を利用する場合は注意してください

インテル® Cilk™ Plus のストランドはオペレーティング・システムのスレッドではありません。1 つのストランドが実行中、ワーカー間で移動されることはありません。しかし、ワーカーは `cilk_spawn`、`cilk_sync` もしくは `cilk_for` ステートメントの後に変わるかもしれません。これらのステートメントは、ストランドを終了したり、生成したりします。さらに、特定のストランドを実行するワーカーを制御する方法もありません。

これによりプログラムは幾つかの重要な影響を受けます：

- Windows のスレッド・ローカル・ストレージ(TLS)や Linux* の Pthread 固有のデータ(TLS など)を利用しないでください。それは、OS スレッドはワーカーがスチールされると変わるかもしれないためです。代わりに、前述した `holder` レデューサーなどを利用してください。
- `cilk_spawn`、`cilk_sync` そして `cilk_for` ステートメントで OS のロックやミューテックスを利用しないでください。これは、ロックしたスレッドだけがオブジェクトを解放できるためです。

Microsoft Foundation クラスとインテル® Cilk™ Plus プログラム

注) このトピックは Windows* プログラマー向けです。

Microsoft Foundation Class (MFC) ライブラリーは、クラスラッパーからオブジェクトの GDI ハンドルをマップするスレッド・ローカル・ストレージに依存します。インテル® Cilk™ Plus のストランドは、特定の OS スレッドで実行することが保証されません。インテル® Cilk™ Plus を利用する並列コードは、安全に MFC 関数を呼び出すことはできません。

MFC ベースのアプリケーションが計算主体のタスクを実行するには、通常次の 2 つのメソッドが利用されます:

- 計算主体のタスクを実行するため、ユーザー・インターフェース(UI)スレッドを生成して演算します。演算スレッドは UI スレッドへ更新をポストし、UI への要求へ答えるため UI スレッドを残しておきます。
- 演算主体のコードを UI スレッドで実行し、UI の更新は直接行います。他の UI 要求を扱うため、“メッセージポンプ”を実行します。

ランタイムシステムはオペレーティング・システムのスレッドを切り替えることができるので、インテル® Cilk™ Plus コードはスレッド・ローカル・ストレージに依存する MFC コードと分離されなければいけません。

MFC プログラムに計算主体のスレッドを追加する:

1. OS API (`_beginthreadex` や `AfxBeginThread`)を利用して演算主体のスレッドを生成します。すべての C++ プログラムはインテル® Cilk™ Plus へ変換し、スレッドとして実行します。main (UI)スレッドでメッセージポンプを実行し Windows メッセージと UI の更新を行うため、演算スレッドとは分離します。
2. UI ウィンドウのハンドル(HWND)を演算スレッドへ渡します。演算スレッドが UI を更新する必要がある場合、`PostMessage` を呼び出すことで UI スレッドへメッセージを送ります。`PostMessage` は、Window ハンドルを作成したスレッドに関連するメッセージをまとめてキューへ送ります。`SendMessage` を利用してはいけません。`SendMessage` は、現在のスレッドで実行され、それは正しい UI スレッドではありません。
3. C++プログラムのアルゴリズムとスレッド管理が正しく動作することを確実にします。
4. 演算スレッドにインテル® Cilk™ Plus を実装します。
5. プログラム終了前に main (UI)スレッドが `WaitForSingleObject()`で演算スレッドの終了を待ちます。

QuickDemo では、インテル® Cilk™ Plus アプリケーションが MFC を利用する例を示します。その他の推奨に次のものがあります。

- main UI スレッドが演算スレッドを生成する場合、main スレッドは演算スレッドの終了を待つべきではありません。生成された演算スレッドはリターンし、メッセージポンプが動作するのを許すべきです。
- 演算スレッドに渡されるデータは、スタック上に置かれていないことを確認してください。もしスタック上にデータがあると、演算スレッドの生成がリターンした時に、データは解放され無効になります。
- 演算スレッドに渡されるデータブロックは、利用し終えたら UI へ完了メッセージ送信する前に演算スレッドが解放しなければいけません。

- CWnd::PostMessage の代わりに PostMessage 関数を利用します。これは、演算スレッドの生成が、インテル® Cilk™ Plus コード中で MFC のスレッド・ローカル変数を利用するのを避けるためです。
- インテル® Cilk™ Plus を利用する短い演算は、演算がブラックボックス化されメッセージポンプを利用したり、他のスレッドと通信しない限り、UI スレッドから直接呼び出すことができます。これは、インテル® Cilk™ Plus を利用するかどうか関係なくインタラクティブではない関数を呼び出すことを許します。

インテル® Cilk™ Plus ランタイム API

この節ではインテル® Cilk™ Plus が提供するランタイム API について説明します。

ランタイムシステム API について

インテル® Cilk™ Plus を利用するプログラムは、ランタイムシステムとライブラリー必要とします。ランタイムシステムは、プログラムから利用可能な幾つかの API 関数を提供します。

インテル® Cilk™ Plus のランタイム関数とクラスを使用するには、`cilk/cilk_api.h` をインクルードします。ヘッダーに定義される API のエントリーポイントは以降に示します。ヘッダーファイルはレデューサーへのインターフェースは定義しません。レデューサーの詳細については、レデューサーの節を参照してください。

`__cilkrts_set_param`

```
int __cilkrts_set_param(const char* name, const char* value);
```

この関数は、インテル® Cilk™ Plus ランタイムシステムの引数を制御します。`name` と `value` の 2 つの文字列引数を対で使用します。

`name` 引数は次のように認識されます：

`nworkers`: `value` 引数では利用するワーカーの数を指定します。値は、10 進数か `0x` や `0` を伴う 16 進数や 8 進数です。この関数が呼び出されない場合、ワーカーの数は `CILK_NWORKERS` 環境変数で定義される数です。`CILK_NWORKERS` が定義されていない場合、デフォルト値は利用可能なプロセッサ・コア数が設定されます。この関数は、`cilk_spawn` や `cilk_for` の前に 1 度だけ呼び出される値が有効となります。

呼び出しが成功するとゼロを返しますが、失敗するとゼロ以外を返します。失敗の原因は、認識できない引数、誤った値、不正な場所での呼び出しなどです。

`__cilkrts_get_nworkers`

```
int __cilkrts_get_nworkers(void);
```

この関数は、インテル® Cilk™ Plus タスクに割り当てられているワーカーの数を返します。そして、`__cilkrts_set_param` で変更できないように値を固定します。シリアル化されたコードで呼ばれると、1 を返します。

ワーカーID は隣接する範囲では必要ありません。ID が指定された場合、`__cilkrts_get_nworkers` の値は大きな値を返すことがあります。

`__cilkrts_get_worker_number`

```
int __cilkrts_get_worker_number (void);
```

`__cilkrts_get_worker_number` 関数は、関数が実行されているインテル® Cilk™ Plus ワーカーを示す小さな整数値が帰ります。

`__cilkrts_get_total_workers`

```
int __cilkrts_get_total_workers (void);
```

ランタイムシステムは一時、アクティブなワーカーより多くのワーカーを割り当てる場合があります。`__cilkrts_get_total_workers` は、アクティブではなく利用されていないワーカーを含む総数を返します。言い換えると、ワーカーが割り当てられるかもしれない最大数値を返します。通常この値は、インテル® Cilk™ Plus タスクに割り当てる実際の値を少し上回る値です。`__cilkrts_get_total_workers()` 個の大きさを持つ配列を生成でき、各要素のインデックスはワーカーID です。また、`__cilkrts_get_total_workers` 関数は、`__cilkrts_set_param` で変更できないようにワーカー数を固定します。シリアル化されたコードから呼び出されると、`__cilkrts_get_total_workers` は 1 を返します。

競合状態を理解する

この節では並列プログラミングで排除しなければならないデータ競合とその排除方法について説明します。

データ競合

競合は並列プログラムにおける主なバグの原因です。

2 つの並列ストランドが同じメモリー位置をアクセスし、少なくとも一方が書き込みを行う場合、決定性競合が起きます。プログラムの結果は、どちらのストランドがメモリーを最初にアクセスすることができたか、に依存します。

データ競合は、決定競合の特殊なケースです。データ競合は、2 つの並列ストランドが、ロックなしで同じメモリーをアクセスし、少なくとも 1 つのストランドが書き込みを行う場合に起こす競合状態です。プログラムの結果は、どちらのストランドが最初にメモリーをアクセスしたかに依存します。

もし並列アクセスがロックで保護されているなら、データ競合は起こりません。しかし、ロックを利用するプログラムは決定論的な結果を生成しません。ロックは、更新中参照できるデータ構造を保護することによって、一貫性を保証しますが、決定論的な結果を保証できません。

次のプログラムを考えてみましょう。

```
int a = 2; // declare a variable that is
          // visible to more than one worker

void Strand1()
{
    a = 1;
}

int Strand2()
{
    return a;
}

void Strand3()
{
    a = 2;
}

int main()
{
    int result;
    cilk_spawn Strand1();
    result = cilk_spawn Strand2();
    cilk_spawn Strand3();
    cilk_sync;
    std::cout << "a = " << a << ", result = "
                << result << std::endl;
}
```

Strand1()、Strand2() そして Strand3() は並列に実行され、結果の最終値はそれらが実行される順番

に依存します。

Strand1()は、Strand2()が読み込む前か、後に値を書き込むため、Strand1() と Strand2() には read/write 競合があります。

Strand3()は、Strand1()が書き込む前か、後に値を書き込むため、Strand3() と Strand1() には write/write 競合があります。

いくつかのデータ競合は最悪ではありません。言い換えると、競合は起こるけれども、プログラムの結果には影響しない場合です。

次の例を見てください：

```
bool bFlag = false;
cilk_for (int i=0; i<N; ++i)
{
    if (some_condition()) bFlag = true;
}
if (bFlag) do_something();
```

プログラムが bFlag 変数に write/write 競合を持ちますが、すべての書き込み操作で同じ値を書き込んでいます。そして、値は cilk_for ループの終わりにある明示的な cilk_sync の直後まで読み取られません。

この例では、データ競合は最悪の結果をもたらしません。ループ反復がどのような順番で実行されても、プログラムが生成する結果は同じです。

データ競合を排除する

競合状態を解決するにはいくつかの方法があります：

- プログラムのバグを直す
- グローバル変数の代わりにローカル変数を利用する
- コードを再構築する
- アルゴリズムを変更する
- レデューサーを利用する
- ロックを利用する

プログラムのバグを直す

qsort-race の例にあるような競合はプログラム・ロジック上のバグです。この例では再帰呼び出し sort をオーバーラップした領域に対し行っているため、同時に同じメモリーを参照するため競合します。解決策はアプリケーション自信を修正するしかありません。

グローバル変数の代わりにローカル変数を利用する

次のプログラムを考えてください:

```
#include <cilk/cilk.h>
#include <iostream>
const int IMAX=5;
const int JMAX=5;
int a[IMAX * JMAX];

int main()
{
    int idx;

    cilk_for (int i=0; i<IMAX; ++i)
    {
        for (int j=0; j<JMAX; ++j)
        {
            idx = i*JMAX + j; // This is a race.
            a[idx] = i+j;
        }
    }
    for (int i=0; i<IMAX*JMAX; ++i)
        std::cout << i << " " << a[i] <<std::endl;
    return 0;
}
```

このプログラムでは、idx 変数で競合が起こります。この変数は cilk_for ループから同時にアクセスされません。idx はループ内部でのみ利用されているため、idx をローカル変数とすることで簡単に解決できます。

```
int main()
{
    // int idx; // Remove global
    cilk_for (int i=0; i<IMAX; ++i)
    {
        for (int j=0; j<JMAX; ++j)
        {
            int idx = i*JMAX + j; // Declare idx locally
            a[idx] = i+j;
        }
    }
    ...
}
```

コードを再構築する

いくつかのケースでは、簡単にコードを書き換えることで競合を排除できます。以下は、前述のプログラムを別の方法で解決する例です:

```
int main()
{
    // int idx; // Remove global
    cilk_for (int i=0; i<IMAX; ++i)
    {
        for (int j=0; j<JMAX; ++j)
        {
            // idx = i*JMAX + j; // Don't use idx
        }
    }
}
```

```

    a[i*JMAX + j] = i+j; // Instead,
                        // calculate as needed
    }
}
...
}

```

アルゴリズムを変更する

競合しないような演算は並列性を制限するため、アルゴリズムのパーティション化は最良の方法の 1 つですが、常に容易で最良とは限りません。

レデューサーを使用する

レデューサーは競合なしのオブジェクト向けに設計され、安全に並列処理で利用できます。詳細はレデューサーの節を参照してください。

ロックを使用する

ロックはデータ競合状態を解決できます。ロックを利用する上での欠点は、「ロックを利用する際の考察」で詳しく述べています。ロックには次に示す幾つかの種類があります：

- インテル® スレッディング・ビルディング・ブロックが提供する `tbb::mutex` オブジェクト
- Windows* や Linux* オペレーティング・システムが提供するロック
- 命令の read-modify-write シーケンスを保護する短いロックを提供するアトミック命令

下記のプログラムは `sum` に競合があります。`sum = sum + i` では `sum` への読み込みと書き込みがあります：

```

#include <cilk/cilk.h>

int main()
{
    int sum = 0;
    cilk_for (int i=0; i<10; ++i)
    {
        sum = sum + i; // THERE IS A RACE ON SUM
    }
}

```

ロックを利用して次のように競合を解決します：

```

#include <cilk/cilk.h>
#include <mutex.h>
#include <iostream>

int main()
{
    tbb::mutex mut;
    int sum = 0;
    cilk_for (int i=0; i<10; ++i)
    {
        mut.lock();
        sum = sum + i; // PROTECTED WITH LOCK
        mut.unlock();
    }
    std::cout << "Sum is " << sum << std::endl;
}

```

```
return 0;  
}
```

この例では、解決策を示しているだけ、ということに留意してください。実際にはレデューサーを利用する方法が最良です。

ロックを利用する

この節では競合を排除する際に利用されるロックについて説明します。

ロックを利用する際の考察

ハードウェアやオペレーティング・システムでは幾つかの同期メカニズムを実装できます。

インテル® Cilk™ Plus は、次のロックメカニズムを認識できます。

- インテル® スレッディング・ビルディング・ブロックでは、クリティカルなコード領域で共有メモリーや他の共有リソースを完全に更新しアクセスするため、`tbb::mutex` を提供します。インテル® Parallel Studio 2011 ツールは、ロックを認識し、`tbb::mutex` で保護されたメモリーアクセスに間して競合を報告しません。`qsort-mutex` の例では、`tbb:mutex` の使い方を解説します。
- Windows* OS の場合：Windows の `CRITICAL_SECTION` オブジェクトは、`tbb:mutex` オブジェクトを同じような機能を提供します。インテル® Parallel Studio 2011 ツールは、`EnterCriticalSection()`、`TryEnterCriticalSection()` や `LeaveCriticalSection()` で保護されたメモリーアクセスに間して競合を報告しません。
- Linux* OS の場合：Posix* Pthread の `mutex(pthread_mutex_t)` も、`tbb:mutex` オブジェクトを同じような機能を提供します。インテル® Parallel Studio 2011 ツールは、`pthread_mutex_lock()`、`pthread_mutex_trylock()` や `pthread_mutex_unlock()` で保護されたメモリーアクセスに間して競合を報告しません。
- インテル® Parallel Studio 2011 ツールは、C/C++プログラマーがコンパイラーのイントリンジックスとして利用するアトミック・ハードウェア命令を認識します。

次のロックに関する用語と事実は有益です：

- 次の用語は入れ替え可能です：“獲得(acquiring)”、“突入(entering)”、“ロックする(locking)”、“ロック(もしくは mutex)”。
- スtrand(スレッド)がロックを取得することを、ロックを持つと言います。
- スtrand自身が所有するロックは、“解放”、“退出”、そして“アンロック”できます。
- 1つのスtrandは、同時に1つのロックを所有します。
- `Tbb:mutex` は、OS ミューテックス操作を利用して実装されています。

並列プログラムにおいて、ロックの衝突はパフォーマンスの問題を生じます。その上、ロックはデータ競合を解決できますが、ロックを利用するプログラムは時として非決定論的です。可能であればロックの利用を避けることを推奨します。

これらの問題は、以降の節で詳しく説明します。

ロックは決定論的競合を引き起こす

リソース(単純な変数やリスト、さらに他のデータ構造など)を保護するため、ロックを適切に利用しても 2 つのストランドが変更する実際の順番は決定論的ではありません。例えば、次のコードは、スポンされる関数の 1 部分ですが、複数のストランドを同時に実行されることを想定します。

```

. . .
// グローバル変数 gv を更新する関数を修正
sm.lock();
Update(gv);
sm.unlock();
. . .

```

複数のストランドはロック sm を取得します。同じ入力であっても 1 つのプログラム実行から次へ gv が更新される順番は異なります。これは非決定論性の原因ですが、データ競合で定義される競合ではありません。

非決定論性は、更新が整数加算などのように交換操作を行うならば、異なる最終結果を引き起こさないかもしれない。しかし、リストの最後に要素を追加するような多くの一般的な操作は、交換ではなく結果はロックが要求される順番に依存します。

デッドロック

デッドロックは 2 つ以上のロックを利用し、異なるストランドが異なる順番でロックを要求することで発生します。2 つ以上のストランドが、お互いに一方のストランドが所有するミューテックスを取得する時、デッドロックに陥る可能性があります。

次のコード例は、2 つの異なるストランドが 1 つのリスト要素を他のリストへ移動しています。この方法では、要素は常に 2 つのリストのどちらかに存在します。L1 と L2 は 2 つのリストで、sm1 と sm2 は 2 つの cilk::mutex オブジェクトでありそれぞれ L1 と L2 を保護します。

```

// コード A   L1 のはじめの要素を L2 の最後へ移動
L2.
sm1.lock();
sm2.lock();
L2.push_back(*L1.begin);
L1.pop_front();
sm2.unlock();
sm1.unlock();
. . .
. . .

```

```

// コード B   L2 のはじめの要素を L1 の最後へ移動
sm2.lock();
sm1.lock();
L1.push_back(*L2.begin);
L2.pop_front();
sm2.unlock();
sm1.unlock();

```

1 つのストランドがコード A を実行して sm1 をロックし、他のストランドでコード B が、コード A が sm2 を取得する前に、sm2 をロックすると、デッドロックに陥ります。どちらのストランドも処理を続行できなくなります。

デッドロックを解決する一般的な方法は、各コードで同じ順番でロックを取得することです。つまり、前述の例では、コード B に最初の 2 行を入れ替えます。相反する順番でロックを解放することはできますが、デッドロックを避けるのに必要な操作ではありません。

ロックの衝突

共有ロックに同時にアクセスするような複数のストランドは、並列動作できません。幾つかのプログラムでは、並列性のパフォーマンスの観点からロックは排除されます。極端な場合、そのようなプログラムは、相当するシングルコア（プロセッサ）のシリアルプログラムより極度に遅く動かすこともできます。可能であれば、レデューサーを利用することを考えてください。

ロックを利用する場合、次のガイドラインを考慮してください：

- 可能な限り短い時間の同期オブジェクトにしてください。ロックを取得、データを更新、ロックを解放が一般的なロックを利用する流れですが、ロックを所有している間異質な操作を行ってはいけません。もしアプリケーションが長い間ロックオブジェクトを所有し続けなければいけない場合、そこは並列化の候補であるか再考の余地があります。またこのガイドラインでは、ロックを要求したストランドは常にロックを解放するのを保証するのを確実にします。
- ロックを取得したら同じスコープレベルでそれを解放します。ロックオブジェクトの要求と解放を切り離して行くと、オブジェクトを所有する期間があいまいで、同期オブジェクトの解放に失敗しデッドロックとなる可能性があります。またこのガイドラインでは、ロックを要求したストランドは常にロックを解放するのを保証するのを確実にします。
- `cilk_spawn` や `cilk_sync` にまたがってロックを所有してはいけません。これは `cilk_for` ループも同様です。詳細な説明は以降の節を参照してください。
- ロックを利用する順番が常に同じであることを確実にすることで、デッドロックを避けます。異なる順番でロックを解放する必要はありませんが、パフォーマンスを改選できます。

ストランド間でロックを持つ

最も確実に簡単な方法は、ストランドの境界をまたがってロックを所有しない事です。兄弟ストランドは同じロックを利用できますが、親と子が同じロックを共有すると問題が起こります。次に潜在的な問題を示します：

- `cilk_spawn` や `cilk_sync` 境界の後に生成されたストランドは、親ストランドと同じ OS スレッドで実行される保証はありません。Windows* OS `CRITICAL_SECTION` などの多くの同期ロックでは、それらを取得したスレッド自身が解放を行わなければいけません。
- `cilk_sync` はインテル® Cilk™ Plus ランタイム同期オブジェクトにアプリケーションを適合します。これは予測できない方法でアプリケーションと対話できます。次の例を考えます：

```
#include <cilk/cilk.h>
#include <mutex.h>
#include <iostream>
void child (tbb::mutex &m, int &a)
{
```

```

    m.lock();
    a++;
    m.unlock();
}

void parent(int a, int b, int c)
{
    tbb::mutex m;
    try
    {
        cilk_spawn child (m, a);
        m.lock();
        throw a;
    }
    catch (...)
    {
        m.unlock();
    }
}

```

ここでは、`cilk_spawn` を含む `try` ブロックの最後に暗黙の `cilk_sync` があります。例外が発生すると、すべての子が完了するまで実行を継続できません。もし親が子の前にロックを取得すると、`catch` ブロックは子が完了しないと実行できず、子はロックを取得するまで待機するためデッドロックに陥ります。ガードやスコープを持つロックは、ガードされたオブジェクトのデストラクターは `catch` ブロックが終了するまで実行されないため、解決策にはなりません。

ここで状況を悪くしているのは、`try` ブロックが可視的でないということです。自明ではない (non-trivial) なデストラクターとしてローカル変数が宣言される構文では、暗黙の `try` ブロックが近辺に配置されます。したがって、プログラムがスポンされる時やロックを要求する時、それはおそらく `try` ブロックにあります。

関数が子によって取得されたロックを所有する場合、その関数はロックを解放する前に例外をスローすべきではありません。しかし、ほとんどの関数は例外をスローしない事を保証できないため、次の規則に従います：

- 子のストランドで取得されるであろうロックを取得しない。言い換えると、ロックは兄弟をロックするが子をロックしないということです。
- 子をロックする必要があるなら、異なる関数でロックを取得し、処理を行い、ロックを解放するコードを配置し、スポンと同じ関数でその関数を呼び出すようにします。
- 親ストランドがロックを取得する必要があるなら、データ構造の中に 1 つ以上のプリミティブ型を設定してから、ロックを解放します。ロックを所有する間のスロー (オーバーロード操作を含む)、スポン、同期を伴う関数呼び出しは、`try` ブロックを配置しない事で、安全性を保つことができます。

インテル® Cilk™ Plus の性能上の考察

この節ではインテル® Cilk™ Plus プログラムの性能に関する考察と最適化について説明します。

インテル® Cilk™ Plus の性能上の考察

並列プログラムには、チューニングと改良のため、多くの性能上の考察と可能性があります。

多くの場合インテル® Cilk™ Plus ランタイムは、ワークスチールと呼ばれるスケジュールのアルゴリズムを用いてプロセッサのリソースを効率良く利用します。ワークスチール・アルゴリズムは、ワーカーが実行中に 1 つのプロセッサ・コア(プロセッサ)から他のコア(プロセッサ)へ移動されるのを最小限にするよう設計されています。

さらに、アルゴリズムはワーカーの数に応じてリニアに空間を利用することを確実にします。つまり、インテル® Cilk™ Plus プログラムが N 個のワーカーで動作する場合、1 個のワーカーが必要とする N 倍のメモリを必要とするということです。

粒度

分割統治は効果的な並列化の戦略であり、大小のサブ・プログラムを最適な組み合わせを生成します。ワーク・スチール・スケジューラーは、極端に大きなもしくは小さなチャンクが無く、効果的にコアに割り当てます。もし作業が大きく数少ないチャンクに分割されると、コアをビジーに保つ十分な並列性が無いかもしれません。またチャンクが小さすぎると、スケジュールのオーバーヘッドが並列性の効果を上回るかもしれません。

cilk_for や cilk_spawn を利用する並列プログラムでは、粒度は問題となるかもしれません。cilk_for を利用する場合、ループの grain size を設定して粒度を制御できます。さらにループが入れ子である場合、演算能力は cilk_for が内側のループ、外側のループもしくは両者のいずれかに利用され、最高の性能を達成できるかどうかで決まります。cilk_spawn を利用する場合、小さなチャンクをスポーンしないように注意してください。cilk_spawn のオーバーヘッドが僅かである場合、スポーンが小さな作業であるとパフォーマンスに問題があります。

シリアルプログラムの最適化

並列化の第一歩は、C/C++のシリアルプログラムが、コンパイラーによる最適化を含む通常の最適化手法が活用されていて性能が高いことを確実にすることです。

シリアルプログラム最適化の重要性を示します。matrix_multiply の例では、キャッシュ・ラインミスを最小にすることを意図してループを構成しています。

```
cilk_for(unsigned int i = 0; i < n; ++i){
    for (unsigned int k = 0; k < n; ++k) {
        for (unsigned int j = 0; j < n; ++j) {
            A[i*n + j] += B[i*n + k] * C[k*n + j];
        }
    }
}
```

乗算パフォーマンスの評価において、ここで示す実装は、2 つのループを入れ替えた(k と j ループ)同じプログラムと比較すると、劇的なパフォーマンスの利点を持ちます。この性能差はシリアルプログラムとインテル® Cilk™ Plus で並列化されたプログラムの両方に当てはまる。matrix の例は同じループ構造です。しかし、この性能向上はすべての構成のシステムで達成できるわけではないことに注意してください。各システムには性能に影響する多くの要素があります。

プログラムのタイミングとセグメント

パフォーマンスを測定してボトルネックを理解しなければいけません。僅かな変更でも時には劇的な性能向上を達成することができます。性能をチューニングする唯一の信頼できる方法は、頻繁に性能を測定することです(望ましくは異なる構成のシステムで)。利用可能なツールやテクニックは活用しますが、最適化が有効かどうかは計測することで明らかになります。

性能計測が誤った判断に導くことがあります。しかし注意して潜在的な異常を意識することが重要です。これら意識しなければならない項目の多くは、シンプルですが見落とされる可能性があります。

- 他のアプリケーションが動作していると、対象のアプリケーションの性能に影響を及ぼします。一般的なデスクトップ・アプリケーションがアイドルしている時でさえ、プロセッサ時間を消費し計測結果に影響します。
- プログラム中に計測ポイントがある場合、計測開始位置を含む他のストランドが並列に動作しているなら、2 つの計測ポイント間で経過時間を計らないようにしなければいけません。
- 動的に動作周波数がスケールするラップトップなどのシステムでは、追加のコアを利用するためワーカー数を増やした場合、予知できない性能結果を招くことがあります。ワーカーが増えアクティブなプロセッサ・コアも増加すると、システムは消費電力を考慮して動作周波数を調整します。その結果全体のピーク性能が低下します。

一般的な性能上の落とし穴

プログラムが十分に並列性を持ち負荷が均等でも性能目標に達しない事があります。その場合、他の党その影響を受けているかもしれません。次に一般的な要因を示します。

- **cilk_for の Grainsize(粒度)の設定**
粒度が大きすぎる場合プログラムの論理的な並列性は低下します。逆に小さすぎる場合は、スポンに依存するオーバーヘッドが並列性を阻害します。インテル® Cilk™ Plus とランタイム・システムは、粒度を求めるデフォルトの計算式を持ちます。多くの場合デフォルト設定で程度に動作しますが、粒度を調整して性能を調査します。
- **ロックの衝突**
ロックは一般的に並列性を阻害し性能に影響を及ぼします。ロックの影響はプロファイル・ツールを利用して観測できます。
- **フォルスシェアリング**
次の節で説明します。
- **アトミック操作**

コンパイラーのイントリンジックス(組み込み関数)で提供されるアトミック操作はキャッシュラインをロックします。従ってこの操作はロック衝突と同様に性能に影響を及ぼします。アトミック操作はキャッシュライン全体をロックするため、フォルスシェアが起こります。

キャッシュ効率とバンド幅

キャッシュ効率はシリアルプログラムの性能に影響します。マルチコア環境で動作する並列プログラムではより重要です。プロセッサ・コアは、プロセッサとメモリー間でデータを高速に転送するのを制限するバスのバンド幅を奪い合います。したがって並列プログラムを設計して実装する際には、キャッシュ効率とデータと空間の局所性を考慮します。「シリアルプログラムを最適化する」節で紹介した `matrix` と `matrix_multiply` の例は、これらの問題を考慮しています。

バスのバンド幅に関する問題を特定するには、システムのコア 1 個あたり 1 つの同じシリアルプログラムを同時に実行します。シリアルプログラムの平均実行時間が、複数実行している内の 1 つのコピーよりはるかに大きければ、プログラムはシステムバス帯域を飽和状態にしています。この原因は、メモリー帯域、ディスクもしくはネットワーク I/O 帯域に限界があるかもしれません。

これらの帯域によるパフォーマンスへの影響はシステムによって固有です。前述の `matrix` プログラムを 2 コア(以降 "S2C" と呼びます)の特定のシステムで実行する場合、並列バージョンはシリアルバージョンよりかなり遅いかも知ません(4.431 秒と 1.1435 秒で実測された)。しかし、16 コアとワーカーを持つ他のシステムでは、並列バージョンのコードはリニアにスピードアップするかもしれません。次に S2C における結果を示します:

```
1) Naive, Iterative Algorithm. Sequential and Parallel.
Running Iterative Sequential version...
  Iterative Sequential version took 1.435 seconds.
Running Iterative Parallel version...
  Iterative Parallel version took 4.431 seconds.
  Parallel Speedup: 0.323855
```

あるシステム上で他のシステム(DRAM 速度、メモリーチャンネル数、キャッシュとページテーブルのアーキテクチャー、そして何個の CPU コアがダイに搭載されているかなど)よりメモリーバンド幅が広いことがあるのには、複雑で予測できない幾つかの理由があります。このような効果があるため、予測できず一貫性のない性能となることを意識できるかもしれません。この状況は並列プログラム固有ですが、インテル® Cilk™ Plus プログラムに限ったことではありません。

フォルスシェア

フォルスシェアはメモリー共有型の並列処理では一般的な問題です。これは、2 つ以上のプロセッサ・コアが同じキャッシュラインを所有するときに起こります。

1 つのコアがメモリーラインを保持するキャッシュラインに書き込みを行うと、他のコアの同じキャッシュラインは無効化されます。他のコアは書き込まれたデータそのものを利用していないかもしれませんが、同じキャッシュラインにある異なる要素を利用します。2 番目のコアは自分が利用していたデータに再びアクセスする前に、キャッシュラインを再読み込みしなければいけません。

キャッシュハードウェアはデータの一貫性を確実にしますが、フォルスシェアが起こると潜在的に高いコストが必要となります。フォルスシェアの問題を見つける最良の方法は、ハードウェアカウンターや他のパフォーマンスツールを利用して、予知できない最終レベルキャッシュミスが増加するかどうか見分けることです。

簡単な例として、配列要素の値をインクリメントする `cilk_for` ループを持つ関数がスポンされる状況を考えてみましょう。コンパイラーがストア命令を生成するよりも値をレジスターに保持してループを最適化するように強制します。

```
volatile int x[32];

void f(volatile int *p)
{
    for (int i = 0; i < 100000000; i++)
    {
        ++p[0];
        ++p[16];
    }
}

int main()
{
    cilk_spawn f(&x[0]);
    cilk_spawn f(&x[1]);
    cilk_spawn f(&x[2]);
    cilk_spawn f(&x[3]);
    cilk_sync;
    return 0;
}
```

配列 `x[]` の要素は 4 バイト幅で、64 バイトキャッシュライン(x86 システムでは)には、16 個のデータが格納されます。ループが完了すると、データ競合が無く正しい結果が生成されます。しかし、個々のストランドが隣接する配列要素を更新するとき、キャッシュラインの衝突が発生し劇的にパフォーマンスを低下させます。例えば、16 コアのシステムで実行してみると、1 つのワーカーでは 16 個のワーカーよりも 40 倍高速に動作します。しかし異なるシステムでは結果は変わります。

メモリー割り当てのボトルネック

Linux* や Windows* システムで利用されるメモリー割り当てシステムは、並列プログラムではボトルネックとなることが知られています。プログラムがシステムのヒープからメモリーを割り当てもしくは解放する時 (`malloc`、`free`、`new` もしくは `delete` を利用する)、ランタイム・ライブラリーはヒープデータ構造の衝突を避けるためミューテックスによるロックを使用します。単一のプロセッサにおいても、このロックはかなり代償を伴います。そして、複数のストランドが同時にメモリーを割り当て/解放すると、ロックの衝突はプログラムの並列性を大幅に損ねます。これはヒープのアロケーターはマルチコア(プロセッサ)にはスケールしないということを意味します。

この問題を解決するには、インテル® スレッディング・ビルディング・ブロック(TBB)・ライブラリーが提供するような、スケラブルなメモリーアロケーターを利用することです。TBB のスケラブル・メモリー・アロケーターは、インテル® Cilk™ Plus プログラムでの利用も推奨されます。

追加情報

次の文献は関連する機能やコンセプトに関する追加情報を提供します。

一般的な情報:

Cilk 実装プロジェクトの Web サイト (<http://supertech.csail.mit.edu/cilkImp.html>) は、MIT Cilk プロジェクトへの入り口です。ここには 3 つの講演を含む、実践論、歴史、そして理論上の基礎などへのリンクをもつプロジェクトの概要 (<http://supertech.csail.mit.edu/cilk/>) があります。インテル® Cilk™ Plus は、MIT での Cilk 言語実装をコンセプトにしています。

The Implementation of the Cilk-5 Multithreaded Language
(<http://supertech.csail.mit.edu/papers/cilk5.pdf>) by Matteo Frigo, Charles E. Leiserson, and Keith H. Randall

The Power of Well-Structured Parallelism
(<http://software.intel.com/en-us/articles/The-Power-of-Well-Structured-Parallelism-answering-a-FAQ-about-Cilk>)

シリアルセマンティクス:

4 Reasons Why Parallel Programs Should Have Serial Semantics
(<http://software.intel.com/en-us/articles/Four-Reasons-Why-Parallel-Programs-Should-Have-Serial-Semantics>)

例:

Are Determinacy-Race Bugs Lurking in YOUR Multicore Application?

(<http://software.intel.com/en-us/articles/Are-Determinacy-Race-Bugs-Lurking-in-YOUR-Multicore-Application>)

Global Variable Reconsidered

(<http://software.intel.com/en-us/articles/Global-Variable-Reconsidered>)

これらのアーティクルは、古いレデューサーの実装を説明していますが、コンセプトは現在でも適用可能です。

競合状態:

What Are Race Conditions? Some Issues and Formalizations

(<http://portal.acm.org/citation.cfm?id=130616.130623>)

by Robert Netzer and Barton Miller.

このドキュメントは確定的な競合に一般的な競合を用いています。データ競合は確定的な競合の特殊ケースです。

N ボディー反復におけるソリューション:

<http://software.intel.com/en-us/articles/A-cute-technique-for-avoiding-certain-race-conditions>,
競合なしで実行できる計算の分配の実装について説明しています。

Making Your Cache Go Further in These Troubled Times

(<http://software.intel.com/en-us/articles/Making-Your-Cache-Go-Further-in-These-Troubled-Times>)

キャッシュ効率のよいアルゴリズムについて触れます。

用語

このプログラマーズガイドで使用する重要な用語をアルファベット順に説明します。

アトミック(atomic)

分割できないもの。任意の時点において、命令シーケンスの命令がまったく実行されていないか、あるいはそのすべての命令が実行されているようにほかのストランドから見える場合、ストランドによって実行されるその命令シーケンスはアトミックです。

マイクロプロセッサ(multiprocessor)

1 つのマルチコアチップに搭載された汎用マルチプロセッサ。

インテル® Cilk™ Plus

並列処理を簡単に表現することができる C/C++ プログラミング言語の拡張セット。

cilk_for

反復を並列で個別に実行することができる for ループを示すキーワード。

cilk_spawn

指定したサブルーチン関数を個別に、呼び出し元と並列で実行できることを示すキーワード。

cilk_sync

cilk_sync の後の文を実行する前に、現在の関数内でスポンされたすべての関数の完了を待機することを示すキーワード。

可換演算(commutative operation)

型 (T) の演算 (op) は、型 T の 2 つのオブジェクト a と b に対して $a \text{ op } b = b \text{ op } a$ の場合、可換であるといえます。整数加算やセットの和集合は可換ですが、文字列結合は可換ではありません。

コンカレント・エージェント(concurrent agent)

その他同様のコンカレント・エージェントと同時にプログラム命令を実行する可能性のあるプロセッサ、プロセス、スレッド、その他のエンティティー。

コア (core)

マルチコアチップにおけるシングル・プロセッサ・ユニット。"プロセッサ" および "CPU" という用語は、業界によっては、"コア" と同義語として使用されます。

CPU

Central Processing Unit。"コア" と同義語です。マルチコアチップの 1 プロセッサを指します。

クリティカル・セクション(critical section)

ロックを保持している間に、ストランドによって実行されるコード領域。

クリティカル・パス長(critical-path length)

スパン(span)を参照

データ競合(data race)

共通ロックを保持していない 2 つ以上の並列ストランドが同じメモリー位置にアクセスし、そのうちの少なくとも 1 つのストランドが書き込みを行う場合に発生する競合状態。決定性競合と比較されます。

デッドロック(deadlock)

2 つ以上のストランドが互いに相手がリソースを解放するのを待機していて、どのストランドも先に進めない状態。

決定論的競合(determinacy race)

2 つの並列ストランドが同じメモリー位置にアクセスして、そのうちのどちらかが書き込みを行う場合に発生する競合状態。

決定論(determinism)

同じ入力で複数回実行した際に、どの実行でもプログラムが同じ動作をすること。通常、決定性プログラムは、デバッグが簡単です。

分散メモリー(distributed memory)

複数のプロセッサ間で分割されているコンピューターのストレージ。分散メモリー・マルチプロセッサとは、プロセッサがリモート・プロセッサのメモリーにアクセスするために、リモート・プロセッサにメッセージを送信する必要があるマシンのことです。共有メモリーと対照的です。

実行時間(execution time)

特定のシステムでプログラムの実行にかかる時間。

フォルスシェア(false sharing)

2 つのストランドが同じキャッシュブロックの異なるメモリー位置にアクセスする場合に発生するキャッシュブロックの競合状態。

グローバル変数(global variable)

すべてのローカルスコープの外側に属する変数。非ローカル変数も参照してください。

ハイパーオブジェクト(hyperobject)

インテル® Cilk™ Plus のランタイムでサポートされている言語構造。それぞれのストランドにハイパーオブジェクトの異なるビューを同時に提供し、複数のストランドが共有変数やデータ構造を個別に更新できるようにします。現在提供されているハイパーオブジェクトはレデューサーのみです。

命令(instruction)

プロセッサにより実行される単一の操作。

リニアなスピードアップ(linear speedup)

プロセッサ数に比例したスピードアップ。完全に直線的なスピードアップも参照してください。

ロック(lock)

リソースへの同時アクセスを制限することでアトミック操作を提供するための同期メカニズム。ロックに関する重要な操作として、取得(ロック)と解放(アンロック)があります。多くのロックは、常に 1 つのストランドだけがロックを保持できる mutex として実装されます。

ロック衝突(lock contention)

複数のストランドが同じロックを奪い合う状態。

マルチコア(multicore)

2 つ以上のプロセッサ・コアを集積した半導体チップ。

マルチプロセッサ(multiprocessor)

複数の汎用プロセッサを搭載したコンピューター。

ミューテックス(mutex)

同時に 1 つのストランドだけが取得できる "相互排他" ロック。同時に 1 つのストランドだけが mutex で保護されたクリティカル・セクションを実行できるようにします。Windows* OS は、CRITICAL_SECTION を含むいくつかの種類のロックをサポートしています。Linux* OS は、Pthreads の pthread_mutex_t objects をサポートしています。

非決定性(nondeterminism)

同じ入力で複数回実行した際に、実行ごとにプログラムの動作が異なること。通常、非決定性プログラムは、デバッグが困難です。

非ローカル変数(nonlocal variable)

使用される関数、メソッド、クラスのスコープの外側に属するプログラム変数。インテル® Cilk™ Plus プログラムでは、`cilk_for` ループの外側にスコープがある変数を表します。

並列ループ(parallel loop)

すべての反復を個別に並列で実行可能な `for` ループ。並列ループを指定するための `cilk_for` キーワード。

並列性(parallelism)

ワークとスパンの比率。プロセッサ数が無制限な場合に、アプリケーションで可能な最大のスピードアップを示します。

完全に直線的なスピードアップ(perfect linear speedup)

プロセッサ数とスピードアップが同じ。直線的なスピードアップも参照してください。

プロセス(process)

自身のアドレス空間を持ち、オペレーティング・システムにより管理されるコンカレント・エージェント。メモリーは、明示的なオペレーティング・システムの呼び出しによってのみプロセス間で共有できます。

プロセッサ(processor)

プロセッサは、プログラムの命令をシーケンシャルに実行するためのロジックを実装します。同義語として "コア" が使用されます。

競合状態(race condition)

非決定性の原因。それぞれのストランドの命令を実行するタイミングや順序によって、同時に実行した計算の結果が異なります。

レシーバー(receiver)

関数呼び出しの結果を受け取る変数。

レデューサー(reducer)

インテル® Cilk™ Plus ランタイムシステムがそれぞれのストランドのビューを結合するために使用する(通常は、結合可能な)`reduce()` 2 項演算子が定義されているハイパーオブジェクト。

応答時間(response time)

ユーザーが入力してから結果を取得するまでの計算の実行にかかる時間。

スケールダウン(scale down)

並列アプリケーション が 1 つまたは少数のプロセッサ数でも効率的に実行できること。

スケールアウト(scale out)

アプリケーションの複数のコピーを多数のプロセッサで効率的に実行できること。

スケールアップ(scale up)

並列アプリケーション が多数のプロセッサ数でも効率的に実行できること。直線的なスピードアップも参照してください。

順序一貫性(sequential consistency)

共有メモリーにおけるコンカレント・エージェントによる操作が、エージェントがそれぞれ実行された場合と矛盾しないような形で、全体的な秩序を保ちながら相互に実行される、並列処理におけるメモリーモデル。

シリアル実行(serial execution)

インテル® Cilk™ Plus プログラムのシリアル化の実行。

シリアル・セマンティクス(serial semantics)

インテル® Cilk™ Plus プログラムの直接化を実行したときの動作。

シリアル化(serialization)

インテル® Cilk™ Plus プログラムからインテル® Cilk™ Plus キーワードを除いた C/C++ プログラム。cilk_spawn と cilk_sync は取り除かれ、cilk_for は通常の for ループに置換されます。シリアル化はデバッグに使用できます。また、変換された C/C++ プログラムは、オリジナルの C/C++ プログラムと同じ動作になります。一部の文献では、"シリアル化" という用語が使用されています。"シリアル・セマンティクス" も参照してください。

共有メモリー(shared memory)

複数のプロセッサ間で共有されているコンピューターのストレージ。共有メモリー・マルチプロセッサとは、各プロセッサが任意のメモリー位置に直接アクセスできるマシンのことです。分散メモリーと対照的です。

スパン(span)

無制限のプロセッサ数で並列プログラムを実行した場合の理論上の最小実行時間です(通信とスケジューリングにかかるオーバーヘッドは考慮されません)。通常、文献では T_0 と表されます。クリティカル・パスの長さとも呼ばれます。

スポーン(spawn)

通常の呼び出しのように関数が戻るのを待たずに、関数を呼び出すこと。呼び出し元は、呼び出した関数と並列で実行を継続できます。cilk_spawn も参照してください。

スピードアップ(speedup)

プログラムを並列で実行した際に、1 つのプロセッサで実行した場合よりもどれだけ速く実行されるかを示す値。スピードアップは、プログラムを P 個のプロセッサで実行したときの実行時間 T_P を 1 個のプロセッサで実行したときの実行時間 T_1 で割った値です。

ストランド(strand)

spawn やその他の並列制御からのスポーン、同期、リターンが含まれていない一連のシリアル命令。

同期(sync)

続行する前に、スポーンされた関数のセットがリターンするのを待機すること。現在の関数は、スポーンされた関数に依存していて、それらと並列に続行することができません。cilk_sync も参照してください。

スレッド(thread)

一連のシリアル命令からなるコンカレント・エージェント。同じジョブのスレッドは、メモリーを共有します。通常、スレッドのスケジューリングは、オペレーティング・システムによって管理されます。

スループット(throughput)

単位時間あたりに実行される演算の数。

ビュー(view)

特定のストランドから見たハイパーオブジェクトの状態。

ワーク(work)

シングル・プロセッサでプログラムを実行した場合の実行時間、 T_1 と表されることもあります。

ワークスチール(work stealing)

それぞれのプロセッサでローカルに並列作業を行い、ローカルの作業がなくなったら、別のプロセッサから作業を移動してくるスケジューリング手法。ワーク・スチール・スケジューラーは、並列処理が十分にある場合、通信や同期にオーバーヘッドがかからないため効率的です。インテル® Cilk™ Plus のランタイムシステムは、ワーク・スチール・スケジューラーを使用します。

ワーカー(worker)

別のワーカーが並列ストランドで命令を実行すると同時に 1 つのストランドの命令を実行する可能性のあるコンカレント・エージェント。ワーカーは、インテル® Cilk™ Plus ランタイムシステムのワーク・スチール・スケジューラーにより管理されます。ワーカーは、オペレーティング・システム・スレッドとして実装されます。