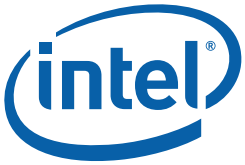


インテル® スレッディング・ビルディング・ブロック

チュートリアル

改訂番号: 1.6

Web サイト: http://www.intel.com

著作権と商標について

本資料に掲載されている情報は、インテル製品の概要説明を目的としたものです。本資料は、明示されているか否かにかかわらず、また禁反言によるとよらずにかかわらず、いかなる知的財産権のライセンスを許諾するためのものではありません。製品に付属の売買契約書『Intel's Terms and Conditions of Sale』に規定されている場合を除き、インテルはいかなる責を負うものではなく、またインテル製品の販売や使用に関する明示または黙示の保証（特定目的への適合性、商品性に関する保証、第三者の特許権、著作権、その他、知的所有権を侵害していないことへの保証を含む）にも一切応じないものとします。インテル製品は、医療、救命、延命措置、重要な制御または安全システム、核施設などの目的に使用することを前提としたものではありません。
インテル製品は、予告なく仕様や説明が変更される場合があります。

機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。

MPEG は、ビデオの圧縮/伸張に関する国際的な規格であり、ISO によって奨励されています。MPEG コーデックまたは MPEG 対応のプラットフォームを実装するには、Intel Corporation をはじめとする各種の団体からライセンスを取得しなければならない場合があります。

本資料で説明されているソフトウェアには、不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのソフトウェアの不具合については、インテルまでお問い合わせください。

本資料およびこれに記載されているソフトウェアはライセンス契約に基づいて提供されるものであり、その使用および複製はライセンス契約で定められた条件下でのみ許可されます。本資料で提供される情報は、情報供与のみを目的としたものであり、予告なく変更されることがあります。また、本資料で提供される情報は、インテルによる確約と解釈されるべきものではありません。インテルは本資料の内容およびこれに関連して提供されるソフトウェアにエラー、誤り、不正確な点が含まれていたとしても一切責任を負わないものとします。

ライセンス契約で許可されている場合を除き、インテルからの文書による承諾なく、本資料のいかなる部分も複製したり、検索システムに保持したり、他の形式や媒体によって転送したりすることは禁じられています。

機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。これらの不正使用により、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。

Intel、インテル、Intel ロゴ、Itanium は、アメリカ合衆国およびその他の国における Intel Corporation の商標です。

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

改訂履歴

バージョン	バージョン情報	日付
1.0	初版。	2006 年 3 月 10 日
1.4	バスタブグラフを 3.2.1 に追加。非ブロック・アルゴリズムをセクション 7 に追加。誤字を修正。テンプレートを更新。3.2 の `operator()` と `const` の関係の論理的根拠を書き換え。	2006 年 12 月 5 日
1.5	TBB_INSTALL_DIR への参照を削除。Partitioner コンセプトをプレビュー機能として追加。	2007 年 3 月 1 日
1.6	ライブラリーのデバッグバージョンとリリースバージョンについて、2.1 にコメントを追加。	2007 年 4 月 18 日

目次

1 はじめに

このチュートリアルでは、インテル® スレッディング・ビルディング・ブロック (インテル® TBB) の使用方法を説明します。インテル® TBB は、スレッド化の専門家の手を借りずに、マルチコアのパフォーマンスを最大限に活用できるよう支援するライブラリーです。最初は難しく感じるかもしれませんが、通常は、何点かのキーポイントを抑えるだけで、マルチコア・プロセッサー用にコードを改良できることがご理解いただけるでしょう。例えば、このドキュメントのセクション 3.4 までを読み進めていくと、プログラムをいくつかスレッド化させることが可能です。徐々に知識が深まると、詳細セクションで記述されているような、より複雑な内容に関心を持たれることでしょう。

1.1 ドキュメントの構成

このチュートリアルは、高レベルな機能、低レベルな機能を順に説明した後、最後に中レベルのタスク・スケジューラーについて説明します。このインテル® スレッディング・ビルディング・ブロックの使用チュートリアルには次のセクションが含まれています。

表 1 ドキュメントの構成

セクション	説明
第 1 章	ドキュメントの概要を説明します。
第 2 章	ライブラリーのインストール方法について説明します。
第 3 章 - 第 4 章	並列ループのテンプレートについて説明します。
第 5 章	コンカレント・コンテナーのテンプレートについて説明します。
第 6 章 - 第 9 章	排他制御、アトミック操作、時間計測、メモリー割り当て用の低レベルの機能について説明します。
第 10 章	タスク・スケジューラーについて説明します。

1.2 利点

並列プログラミングには、プラットフォーム依存のスレッド化プリミティブの使用から、非標準の新しい言語の使用まで、さまざまな手法があります。インテル® スレッディング・ビルディング・ブロックの利点は、ロースレッドよりも高いレベルで動作し、非標準言語やコンパイラーを必要としない点にあります。インテル® スレッディング・ビルディング・ブロックは、ISO C++ をサポートしている任意のコンパイラーで使用できます。このライブラリーは、次の点で典型的なスレッド化パッケージとは異なります。

- **インテル® スレッディング・ビルディング・ブロックでは、スレッドの指定ではなく、*タスク*の指定が可能。**たいていのスレッド化パッケージでは、スレッドの指定が要求されます。スレッドは、ハードウェアに非常に近い、低レベルで重要な構造体であるため、スレッドを直接プログラミングすることは骨の折れる作業であり、また非効率的なプログラムが作成される可能性も高くなります。スレッドを直接プログラミングする場合は、論理タスクをスレッドに効率的にマッピングする必要があります。しかし、インテル® スレッディング・ビルディング・ブロックのランタイム・ライブラリーを使用すれば、プロセッサー・リソースを効率的に活用できるように、自動的にタスクがスレッドにスケジューリングされます。
- **インテル® スレッディング・ビルディング・ブロックは、*パフォーマンスに特化したスレッド化*が対象。**汎用を目的としたスレッド化パッケージの多くは、グラフィカル・ユーザー・インターフェイスにおける非同期イベントのスレッド化など、さまざまなスレッド化をサポートしています。その結果、ソリューションではなくファウンデーションを提供する低水準のツールとなる傾向があります。一方、インテル® スレッディング・ビルディング・ブロックは、計算負荷の高い作業の並列化を目的としているため、より高水準で、より簡単なソリューションを提供します。
- **インテル® スレッディング・ビルディング・ブロックは、その他のスレッド化パッケージと*互換性*を確保。**ライブラリーは、すべてのスレッド化問題に対応するようには設計されていないため、その他のスレッド化パッケージともシームレスに共存できるようになっています。
- **インテル® スレッディング・ビルディング・ブロック は、*スケーラブルなデータ並列プログラミング*に重点。**プログラムをそれぞれ機能別のブロックに分け、各ブロックに個別のスレッドを割り当てる方法では、機能別のブロック数が固定されてしまうため、パフォーマンスの向上はあまり見込めません。一方、インテル® スレッディング・ビルディング・ブロックの場合、*データ並列*プログラミングに重点が置かれているため、コレクションのあらゆる部分をマルチスレッドで動作させることができます。データ並列プログラミングは、コレクションを小さなまとまりに分

割することにより、より多くのプロセッサー数にも対応します。データ並列プログラミングにより、プロセッサーを追加するごとにプログラムのパフォーマンスを向上させることができます。

- **インテル® スレッディング・ビルディング・ブロックは、*汎用プログラミング*** *を使用。*従来のライブラリーでは、特定の型や基本クラスでインターフェイスを指定します。しかし、インテル® スレッディング・ビルディング・ブロックでは、汎用プログラミングを使用します。汎用プログラミングの本質は、最小限の制約で最良のアルゴリズムを作成する点にあります。C++ 標準テンプレート・ライブラリー (STL) は、インターフェイスが型の*要件*によって指定される汎用プログラミングの良い例です。例えば、C++ STL には、シーケンス上のイテレーターで任意に定義されるシーケンスをソートするテンプレート関数 `sort` があります。イテレーターの要件は次のとおりです。

1. ランダムアクセスを提供すること。
2. 式 `*i<*j` は、イテレーター `j` によってポイントされたアイテムの前に配置された イテレーター `i` によりアイテムがポイントされた場合は true で、そうでない場合は false であること。
3. 式 `swap(*i,*j)` は、2 つの要素を交換すること。

型の要件に関する仕様では、テンプレートを使用して、ベクトルやデックといった、あらゆるシーケンス表現をソートすることができます。同様に、インテル® スレッディング・ビルディング・ブロックのテンプレートは、特定の型ではなく、型の要件を指定するため、さまざまなデータ表現に応用することが可能です。汎用プログラミングにより、インテル® スレッディング・ビルディング・ブロックでは、幅広く応用することができるハイパフォーマンス・アルゴリズムを提供します。

2 パッケージの内容

インテル® スレッディング・ビルディング・ブロックには、動的共有ライブラリー・ファイル、ヘッダーファイルが含まれています。また、この章の説明に従ってコンパイルして実行することができる Windows* システム、Linux* システム、Mac OS* システム用のコードサンプルも提供されています。

2.1 デバッグ・ライブラリーとリリース・ライブラリー

インテル® スレッディング・ビルディング・ブロックには、表 2 で説明されているように、デバッグバージョンとリリースバージョンの動的共有ライブラリーが含まれています。

表 2: インテル® スレッディング・ビルディング・ブロックに含まれている動的共有ライブラリー

ライブラリー (*.dll、lib*.so または lib*.dylib)	説明	使用方法
tbb_debug **tbbmalloc_debug**	**ライブラリーの誤用に対する広範囲な内部チェック機能を備えています。**	**1 に設定されたマクロ** TBB_DO_ASSERT **でコンパイルされるコードとともに使用します。**
tbb **tbbmalloc**	**トップパフォーマンスを提供します。ライブラリーの使用に関するほとんどの確認作業を排除します。**	**未定義またはゼロに設定された** TBB_DO_ASSERT **でコンパイルされるコードとともに使用します。**

ヒント: 最初にデバッグバージョンのライブラリーでプログラムをテストして、適切にライブラリーが使用されているかどうかを確認します。リリースバージョンでは、ライブラリーが適切に使用されていない場合、予測できないプログラム動作が生じることがあります。

すべてのバージョンのライブラリーでインテル® スレッドチェッカーとインテル® スレッド・プロファイラーがサポートされています。デバッグバージョンでは、常にフルサポートが有効です。リリースバージョンでは、TBB_DO_THREADING_TOOLS マクロを 1 にしてフルサポートを有効にしてコードをコンパイルする必要があります。

注意: インテル® スレッドチェッカーのインストルメンテーション・サポートは、タスク・ライブラリー (3.1) が最初に初期化された後で有効になります。この初期化が行われる前にライブラリー・コンポーネントが使用されると、実際には競合状態が発生していない場合でも、インテル® スレッドチェッカーが誤って競合状態をレポートする可能性があります。

2.2 スケーラブル・メモリー・アロケーター

インテル® スレッディング・ビルディング・ブロックのデバッグバージョンとリリースバージョンは、一般的なサポートとスケーラブル・メモリー・アロケーターの 2 つの動的共有ライブラリーに分けられます。後者は、名前に malloc が付きます。例えば、Windows システムのリリースバージョンはそれぞれ tbb.dll と tbbmalloc.dll です。アプリケーションでは、一般ライブラリーのみを選択するもの、スケーラブル・メモリー・アロケーターを選択するもの、または両方を選択するものがあります。セクション 9.1 では、インテル® スレッディング・ビルディング・ブロックのどの部分が、どのライブラリーに依存するかを説明します。

2.3 Windows システム

Windows システムのデフォルトのインストール・ディレクトリーは、表 3 で説明されているように、ホスト・アーキテクチャーに依存します。

表 3: インテル® スレッディング・ビルディング・ブロックのデフォルトのインストール先

ホスト	デフォルトのインストール先
インテル® IA-32 プロセッサー	C:\Program Files\Intel\TBB\x.x\
インテル® 64 対応プロセッサー	C:\Program Files (x86)\Intel\TBB\x.x\

x.x には TBB のリリースバージョンが適用されます。

表 4 は、Windows システムのサブディレクトリーの内容についての説明です。

表 4: インテル® スレッディング・ビルディング・ブロックのサブ・ディレクトリー (Windows)

アイテム	場所	環境変数
インクルード・ファイル	include\tbb*.h	INCLUDE
.lib ファイル	<*arch*>\vc<*vcversion*>\lib\<*lib*>.lib	LIB

<table>
<tr>
<td>.dll ファイル</td>
<td><arch>\vc<vcversion>\bin\<lib>malloc>.dll
<arch>:
<table>
<tr><th><arch></th><th>プロセッサー</th></tr>
<tr><td>ia32</td><td>インテル® IA-32 プロセッサー</td></tr>
<tr><td>em64t</td><td>インテル® 64 対応プロセッサー</td></tr>
</table>
<vcversion>:
<table>
<tr><th><vcversion></th><th>環境</th></tr>
<tr><td>7.1</td><td>Visual Studio* .NET 2003</td></tr>
<tr><td>8</td><td>Visual Studio 2005</td></tr>
</table>
<<i>lib</i>>:
<table>
<tr><th><lib></th><th>バージョン</th></tr>
<tr><td>tbb.dll</td><td>リリースバージョン</td></tr>
<tr><td>tbb_debug.dll</td><td>デバッグバージョン</td></tr>
</table>
<malloc>:
<table>
<tr><th><malloc></th><th>バージョン</th></tr>
<tr><td>(なし)</td><td>一般ライブラリー</td></tr>
<tr><td>malloc</td><td>メモリー・アロケーター</td></tr>
</table></td>
<td>PATH</td>
</tr>
<tr>
<td>例</td>
<td>examples\class*\.</td>
<td></td>
</tr>
<tr>
<td>Visual Studio ソリューション・ファイル (サンプル)</td>
<td>examples\class*\vcvcversion*.sln
class は、デモされるクラス、vcversion は、.dll ファイルです。</td>
<td></td>
</tr>
</table>

最後の列は、これらのサブディレクトリーを認識するために Microsoft* コンパイラーまたはインテルのコンパイラーによって使用される環境変数です。

注意: 適切な製品ディレクトリーが環境変数によって指定されるようにしてください。適切に指定されない場合、コンパイラーが必要なファイルを認識できないことがあります。

注意: Windows ランタイム・ライブラリーは、スレッドセーフな形式とスレッドセーフではない形式で提供されます。インテル® スレッディング・ビルディング・ブロックでスレッドセーフではないバージョンを使用すると、未定義の結果を生じることがあります。インテル® スレッディング・ビルディング・ブ

ロックを使用する場合は、スレッドセーフなバージョンを使用していることを確認してください。表 5 は、`cl` または `icl` を使用する場合の必要なオプションを示しています。

表 5: スレッドセーフな C/C++ ランタイムのバージョンをリンクするコンパイラー・オプション

オプション	説明
/MDd	**スレッドセーフなランタイムのデバッグバージョン**
/MD	**スレッドセーフなランタイムのリリースバージョン**

これらのオプションのいずれかを使用しないと、ビルド中にエラーが報告されることがあります。

2.3.1 Visual Studio のコードサンプル

`examples\*\*\` にあるソリューション・ファイルの 1 つを実行するには:

1. `vc7.1` ディレクトリー (Visual Studio .NET 2003 の場合) または `vc8` ディレクトリー (Visual Studio 2005 の場合) を開きます。
2. `.sln` ファイルをダブルクリックします。
3. Visual Studio で、**ctrl + F5** を押し、サンプルをコンパイルして実行します。**Ctrl + F5** (**Shift + F5** ではない) を使用すると、サンプルの終了後にコンソールウィンドウを確認できます。

Visual Studio ソリューション・ファイルのサンプルでは、環境変数はライブラリーがインストールされた場所を指定していなければなりません。インストーラーで、この変数が設定されます。

サンプルの makefile では、`INCLUDE`、`LIB`、`PATH` を 表 4 で示されているように設定する必要があります。次のいずれかの方法で `INCLUDE`、`LIB`、`PATH` を設定することを推奨します。

ヒント: インストーラーの実行時に [Register environment variables (環境変数の登録)] ボックスをオンにします。

または、ライブラリーの `<arch>\vc<vcversion>\bin\` ディレクトリーの `tbbvars.bat` バッチファイルを実行します。`<arch>` と `<vcversion>` については表 4 の説明のとおりです。

2.4 Linux システム

Linux システムでは、デフォルトのインストール先は、/opt/intel/tbb/x.x/ (x.x は TBB のリリースバージョン) です。表 6 は、サブディレクトリーの説明です。

表 6: インテル® スレッディング・ビルディング・ブロック・サブディレクトリー (Linux)

<table>
<tr><th>アイテム</th><th>場所</th><th>環境変数</th></tr>
<tr><td>インクルード・ファイル</td><td>include/tbb/*.h</td><td>CPATH</td></tr>
<tr><td>共有ライブラリー</td><td><arch>/cc<gccversion>_libc<glibcversion>_kernel<kernelversion>/lib/<lib><lib><malloc>.so
各アイテムの意味は次のとおりです。
<table>
<tr><th><arch></th><th>プロセッサー</th></tr>
<tr><td>ia32</td><td>インテル® IA-32 プロセッサー</td></tr>
<tr><td>em64t</td><td>インテル® 64 対応プロセッサー</td></tr>
<tr><td>itanium</td><td>インテル® Itanium® プロセッサー</td></tr>
</table>
<table>
<tr><th><*version> 文字列</th><th>Linux 構成</th></tr>
<tr><td><gccversion></td><td>gcc バージョン番号</td></tr>
<tr><td><glibcversion></td><td>glibc.so バージョン番号</td></tr>
<tr><td><kernelversion></td><td>Linux カーネルバージョン番号</td></tr>
</table>
<table>
<tr><th><lib></th><th>バージョン</th></tr>
<tr><td>tbb.so</td><td>リリースバージョン</td></tr>
<tr><td>tbb_debug.so</td><td>デバッグバージョン</td></tr>
</table>
<table>
<tr><th><malloc></th><th>バージョン</th></tr>
<tr><td>(なし)</td><td>一般ライブラリー</td></tr>
<tr><td>malloc</td><td>メモリー・アロケーター</td></tr>
</table></td><td>LIBRARY_PATH
LD_LIBRARY_PATH</td></tr>
<tr><td>サンプル</td><td>examples/<class>/*/</td><td></td></tr>
<tr><td>サンプル用 GNU Makefile</td><td>examples/<class>/*/linux/Makefile
class は、デモされるクラスです。</td><td></td></tr>
</table>

2.5 Mac OS システム

Mac OS の場合、ライブラリーのデフォルトのインストール先は、`/Library/Frameworks/Intel_TBB.framework/Versions/x.x/` (`x.x` は TBB のリリースバージョン) です。表 7 は、サブディレクトリーの説明です。

表 7: インテル® スレッディング・ビルディング・ブロック サブディレクトリー (Mac OS X)

<table>
<tr><th>アイテム</th><th>場所</th><th>環境変数</th></tr>
<tr><td>インクルード・ファイル</td><td>include/tbb/*.h</td><td>CPATH</td></tr>
<tr><td>共有ライブラリー</td><td>ia32/cc<<i>gccversion</i>>_os<<i>osversion</i>>/lib/<<i>lib</i>><<i>malloc</i>>.dylib 説明:
<table>
<tr><th><*version> 文字列</th><th>OS/X 構成</th></tr>
<tr><td><gccversion></td><td>gcc バージョン番号</td></tr>
<tr><td><osversion></td><td>Mac OS/X バージョン番号</td></tr>
</table>
<table>
<tr><th><lib></th><th>バージョン</th></tr>
<tr><td>libtbb.dylib</td><td>リリースバージョン</td></tr>
<tr><td>libtbb_debug.dylib</td><td>デバッグバージョン</td></tr>
</table>
<table>
<tr><th><malloc></th><th>バージョン</th></tr>
<tr><td>(なし)</td><td>一般ライブラリー</td></tr>
<tr><td>malloc</td><td>メモリー・アロケーター</td></tr>
</table></td><td>LIBRARY_PATH
DYLD_LIBRARY_PATH</td></tr>
<tr><td>例</td><td>examples/<<i>class</i>>/*/</td><td></td></tr>
<tr><td>サンプル用 GNU Makefile</td><td>examples/<<i>class</i>>/*/mac/Makefile
<i>class</i> は、デモされるクラスです。</td><td></td></tr>
</table>

3 単純なループの並列化

最も単純な形のスケーラブルな並列処理は、それぞれが干渉することなく同時に実行することのできるループの反復です。次のセクションでは、単純なループの並列化について説明します。

3.1 ライブラリーの初期化と終了

ライブラリーからのアルゴリズム・テンプレートを使用するスレッドやタスク・スケジューラーでは、tbb::task_scheduler_init オブジェクトが初期化されている必要があります。

メモ: インテル® スレッディング・ビルディング・ブロックのコンポーネントは、tbb 名前空間で定義されています。名前空間でコンポーネントの最初の宣言は明示的に指定されますが、その後は暗黙的です。

スレッドには、同時に初期化されたこれらのオブジェクトのうちの複数が含まれていることがあります。オブジェクトは、タスク・スケジューラーが必要なタイミングを示します。タスク・スケジューラーは、すべての task_scheduler_init オブジェクトが終了した時点でシャットダウンされます。デフォルトでは、task_scheduler_init のコンストラクターが初期化を行い、デストラクターが終了処理を行います。次のように、main() で task_scheduler_init を宣言すると、両方でスケジューラーが開始、終了されます。

```
#include "tbb/task_scheduler_init.h"
using namespace tbb;

int main() {
    task_scheduler_init init;
     ...
    return 0;
}
```

付録 B は、プログラムが別のインターフェイスを使用してスレッドを作成する場合の task_scheduler_init オブジェクトの構築方法を説明しています。

サンプルにある using 宣言子は、各識別子の前に名前空間のプリフィックスの tbb を付けなくてもライブラリー識別子を使用できるようにします。サンプルの残りの部分では、using 宣言子が存在していると仮定されます。

task_scheduler_init のコンストラクターには、呼び出し元スレッドを含む、スレッド数を指定するオプションのパラメーターを指定できます。オプションのパラメーターは、次のいずれかです。

- task_scheduler_init::automatic 値。この値は、パラメーターをまったく指定しない場合と同じです。これは、task_scheduler_init::initialize メソッド用です。
- task_scheduler_init::deferred 値。この値は、task_scheduler_init::initialize(n) メソッドが呼び出されるまで初期化を保留します。*n* 値は、コンストラクターのオプションのパラメーターの有効な値です。
- 使用するスレッド数を指定する正の整数。引数は、開発中のスケーリングを調査する場合にのみ指定します。製品コードの場合は、パラメーターを省略するか、または task_scheduler_init::automatic を使用します。

別の task_scheduler_init がアクティブな場合、このパラメーターは無視されます。つまり、スレッド数の不整合は、最初の task_scheduler_init でスレッド数を指定することで解決されます。製品コードでスレッド数を指定しない理由は、大規模なソフトウェア・プロジェクトでは、さまざまなコンポーネントでほかのスレッド向けに最適化できるスレッド数を知る方法がないためです。ハードウェア・スレッドは、共有グローバルリソースです。そのため、タスク・スケジューラーにスレッド数の指定を任せるのが最良の方法です。

ヒント: スレッド数以上のタスクを作成して、タスク・スケジューラーにタスクからスレッドへのマッピングを選択させるようにプログラムを設計してください。

task_scheduler_init が破棄される前にライブラリーを終了する task_scheduler_init::terminate メソッドがあります。次の例は、スケジューラーで使用するスレッド数の決定を遅らせ、ライブラリーを早めに終了します。

```
int main( int argc, char* argv[] ) {
    int nthread = strtol(argv[0],0,0);
    task_scheduler_init init(task_scheduler_init::deferred);
    if( nthread>=1 )
      init.initialize(nthread);
  ... code that uses task scheduler only if nthread>=1 ...
    if( nthread>=1 )
        init.terminate();
    return 0;
}
```

上のサンプルでは、terminate() への呼び出しを省略することができます。これは、task_scheduler_init のデストラクターが task_scheduler_init が初期化されかどうかを確認し、初期化されていれば、終了処理を行うためです。

ヒント: タスク・スケジューラーは、起動時とシャットダウン時に多少負荷がかかるため、並列アルゴリズム・テンプレートを使用するたびにスケジューラーが作成されないように、main に task_scheduler_init を配置してください。

3.2 parallel_for

Foo 関数を配列の各要素に適用させ、各プロセスを同時に処理しても安全だと仮定します。コードは次のようになります。

```
void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i<n; ++i )
        Foo(a[i]);
}
```

ここでの反復空間は、size_t 型で、0 から n–1 を移動します。テンプレート関数 tbb::parallel_for は、反復空間をチャンクに分け、個別のスレッド上でそれぞれのチャンクを実行します。このループの並列化における最初のステップは、ループ本体をチャンクで動作するような形式に変換することです。この形式とは、*ボディー・オブジェクト*と呼ばれる STL 式の関数オブジェクトで operator() がチャンクを処理します。次のコードは、ボディー・オブジェクトを宣言します。インテル® スレッディング・ビルディング・ブロックに必要なコードは、青で示されています。

```
#include "tbb/blocked_range.h"

class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<size_t>& r ) const {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
           Foo(a[i]);
    }
    ApplyFoo( float a[] ) :
        my_a(a)
    {}
};
```

operator() への引数に注目してください。blocked_range<T> は、ライブラリーによって提供されるテンプレート・クラスです。これは、型 T を超える 1 次元の反復空間を示します。parallel_for クラスは、ほかの種類の反復空間でも動作します。ライブラリーは、2 次元空間に blocked_range2d を提供します。セクション 3.4 で説明されるように独自の空間を定義することもできます。

ApplyFoo のインスタンスでは、オリジナルのループの外に定義され、ループ内で使用されるすべてのローカル変数を認識するメンバーフィールドが必要です。通常、ボディー・オブジェクトのコンストラクターはこれらのフィールドを初期化しますが、parallel_for は、ボディー・オブジェクトが作成される方法は問いません。parallel_for テンプレート関数では、ボディー・オブジェクトに、各ワーカースレッドの個別のコピー (または複数のコピー) を作成するために起動されるコピー・コンストラクターが必要です。また、コピー・コンストラクターはこれらのコピーを破棄するデストラクターも起動します。ほとんどの場合、暗黙的に生成されたコピー・コンストラクターとデストラクターは正常に動作します。正常に動作しない場合は、(C++ のように) *両方*が一貫性があるように定義していない場合がほとんどです。

ボディー・オブジェクトはコピーされることがあるため、その operator() では、ボディーは変更できません。変更された場合は、operator() がオリジナルとコピーのどちらで動作するかによって、parallel_for を起動するスレッドに対して可視性があったり、なかったりします。この意味合いを裏付けるものとして、parallel_for では、ボディー・オブジェクトの operator() が const として宣言される必要があります。

サンプルの operator() は、my_a をローカル変数の a にロードします。この処理は必須ではありませんが、サンプルでは、次の 2 つの理由により、この処理を行っています。

- **スタイル**。ループ本体をよりオリジナルに近付けます。
- **パフォーマンス**。ローカル変数はコンパイラーによりトラッキングされやすいため、頻繁にアクセスされる値をローカル変数に置くことで、コンパイラーによるループの最適化に、より効果が見込めます。

ループ本体をボディー・オブジェクトとして記述してしまえば、次のように parallel_for テンプレート関数が起動されます。

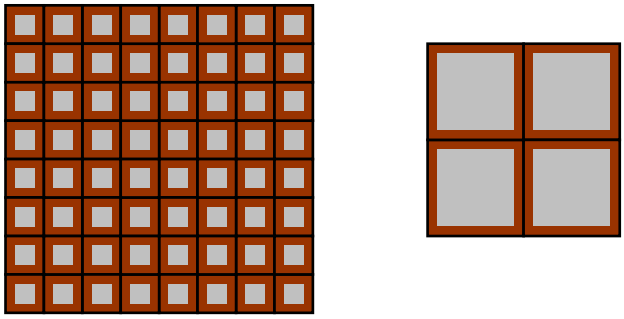
```
#include "tbb/parallel_for.h"

void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n,IdealGrainSize), ApplyFoo(a) );
}
```

ここで構築されている blocked_range は、0 から n-1 までの反復空間全体を表現し、parallel_for は、各プロセッサー用にサブ空間に分割します。コンストラクターの一般的な形式は、blocked_range<T>(*begin*,*end*,*grainsize*) です。*T* は、値の型を指定します。引数の *begin* と *end* は、反復空間の STL 形式を半開区間 [*begin*,*end*) として指定します。

3.2.1 grainsize

3 つ目の引数、*grainsize* は、プロセッサーに分配する “妥当なサイズ” のチャンクに割り当てる反復回数を指定します。反復空間が *grainsize* の反復回数よりも多い場合、`parallel_for` により、個別にスケジューリングされるサブ範囲に反復空間が分割されます。

grainsize により、余分な並列オーバーヘッドを回避することが可能です。並列ループ構造では、それぞれのサブ範囲でオーバーヘッドが生じます。サブ範囲が小さすぎると、有用な作業を超えるオーバーヘッドが生じる可能性があります。grainsize を指定することで、このオーバーヘッドが制限され、並列化の最小しきい値が効率良く設定されます。



ケース A　　ケース B

図 1: オーバーヘッドのパッケージングと grainsize

図 1 は、有用な作業を茶色の箱の中の灰色の領域で表示して、オーバーヘッドの影響を示したものです。ケース A もケース B も、灰色の量は同じです。ケース A は、grainsize が小さすぎたため、比較的大きな割合でオーバーヘッドが生じている様子を示しています。ケース B は、grainsize を 大きく指定したことにより、潜在的な並列処理の機会を犠牲にして、オーバーヘッドの割合を小さくした様子を示しています。有用な作業の一部としてのオーバーヘッドは、粒度の数ではなく、grainsize に依存します。grainsize を設定する際には、反復回数の合計やプロセッサー数ではなく、この関係を考慮してください。

大体、`operator()` の `grainsize` には、少なくとも 10,000 から 100,000 命令の実行が必要です。次の操作を行って確かめることができます。

1. `grainsize` パラメーターを必要な値よりも高く設定します。それぞれのループの反復には、反復ごとに少なくとも、いくつかの命令が必要なため、通常は 10,000 を開始値として設定してください。
2. アルゴリズムを 1 つのプロセッサーで実行します。

3. grainsize パラメーターを半分にしていき、値が少なくなるにつれ、どのくらいアルゴリズムの速度が遅くなるかを確認します。

単一スレッドで実行したときに、5- から 10% ほど速度が遅くなる程度がたいていの目的で最適な設定です。grainsize の値を高く設定することの短所は、並列処理を減少させてしまうことです。例えば、grainsize が 10,000 でループに 20,000 回の反復がある場合、`parallel_for` は、プロセッサーが 2 つ以上ある場合でも、ループを 2 つのプロセッサーにしか分配しません。ただし、必要な値がわからない場合は、低すぎる値よりも少々高い値を設定した方が良いでしょう。値が低すぎると直列パフォーマンスが損なわれ、コールツリーの上位で利用可能な並列処理がある場合には、同様に並列パフォーマンスにも影響をおよぼします。

ヒント: grainsize は厳密に設定する必要はありません。

図 2 は、実行時間と grainsize の関係を示した典型的な "バスタブ曲線" です。100 万個以上のインデックスに対する浮動小数点計算 `a[i]=b[i]*c` に基づいています。反復ごとに、少量の作業があります。時間は 8 個のハードウェア・スレッドを備えた 4 ソケットのコンピューターで測定されています。

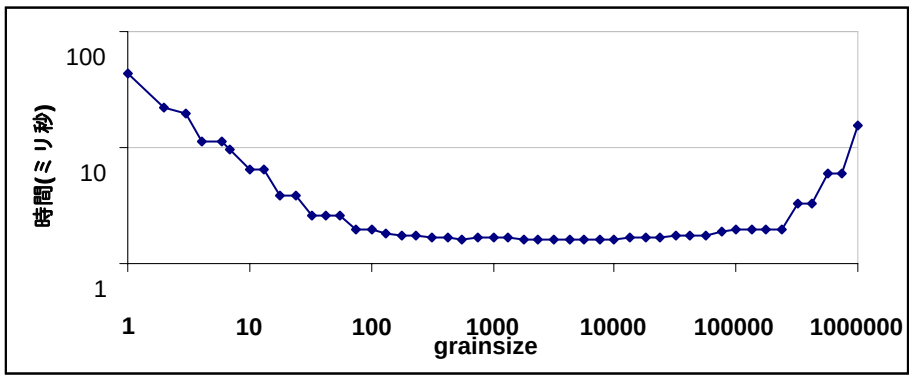


図 2: ウォールクロック時間と grainsize

目盛りは、対数目盛りを使用しています。左側の下降曲線は、grainsize が 1 の場合、ほとんどのオーバーヘッドが並列スケジューリングのオーバーヘッドで、有用な作業ではないことを示しています。grainsize が増えるにつれ、並列オーバーヘッドの割合が減少しています。その後、grainsize の大きさが十分になり、並列オーバーヘッドがわずかになるため、曲線は、平らになります。一番右のグラフの終点では、チャンクが大きすぎて、利用可能なハードウェア・スレッドよりもチャンクが少なくなるため、曲線は上向きます。100 から 100,000 の範囲で grainsize の効果があることに注目してください。

ヒント: ループの入れ子を並列化する場合は、最外ループを可能な限り並列化すると良いでしょう。これは、外ループの各反復が、内ループの繰返しに比べ、より大きな粒度の作業を行う傾向があるためです。

3.2.2 帯域幅

単純な関数 Foo では、並列ループで記述された場合に、大幅な速度向上が達成されない可能性があります。これは、プロセッサーとメモリー間の不十分なシステム帯域幅が原因です。この場合、キャッシュを活用することを考えてアルゴリズムを検討する必要があります。キャッシュの利用を考慮した構造の見直しは、並列プログラムやシリアルプログラムに有益です。

3.2.3 パーティショナーの使用

インテル® スレッディング・ビルディング・ブロックのバージョン 1.1 では、パーティショナーがサポートされています。パーティショナーは、範囲をチャンクに分割する指針を担うオブジェクトです。`simple_partitioner` と `auto_partitioner` の 2 つのパーティショナー・クラスから選択が可能です。

`simple_partitioner` は、3.2.1 で説明されているデフォルトポリシーを実装します。これは、`parallel_for` (3.4)、`parallel_reduce` (3.5)、`parallel_scan` (3.6) のデフォルトです。`simple_partitioner` には、次の利点があります。

1. 概念が単純である。
2. operator() のサブ範囲の最大サイズを想定できるため、チャンクが grainsize 反復回数を超えない。これにより、operator() は、動的な一時的配列の代わりに、固定サイズの一時的な配列を使用することができます。
3. 特定のコンピューター向けのチューニングを行うことができる。

`simple_partitioner` の短所は、3.2.1 の説明にあるように、適切な grainsize を見つけることが必要なこと、そして最適な grainsize は、コンピューター固有である可能性が高いということです。

`auto_partitioner` は、ヒューリスティックにチャンクサイズを選択することにより、grainsize を指定する必要がありません。ヒューリスティックでは、負荷のバランスがとれるよう十分な機会を提供しながら、オーバーヘッドを抑制します。

以下のコードは、grainsize の代わりに、auto_partitioner を使用する方法を示します。blocked_range が構築される際、auto_partitioner オブジェクトが parallel_for の 3 つ目の引数として渡され、grainsize パラメーターが省略されていることに注意してください。

```
#include "tbb/parallel_for.h"

void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n), ApplyFoo(a),
                 auto_partitioner() );
}
```

ほとんどのヒューリスティックでは、auto_partitioner で最適な判断が下せないために、simple_partitioner の方がパフォーマンスが向上する場合があります。auto_partitioner は、調査する時間があって、対象のコンピューターの grainsize をチューニングすることができる場合に使用することを推奨します。

3.3 parallel_reduce

以下のように、ループでリダクションを行うことができます。

```
float SerialSumFoo( float a[], size_t n ) {
    float sum = 0;
    for( size_t i=0; i!=n; ++i )
        sum += Foo(a[i]);
    return sum;
}
```

反復が独立している場合、次のように parallel_reduce テンプレート・クラスを使用してこのループを並列化することができます。

```
class SumFoo {
    float* my_a;
public:
    float sum;
    void operator()( const blocked_range<size_t>& r ) {
        float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i )
            sum += Foo(a[i]);
    }

    SumFoo( SumFoo& x, split ) : my_a(x.my_a), sum(0) {}

    void join( const SumFoo& y ) {sum+=y.sum;}

    SumFoo(float a[] ) :
        my_a(a), sum(0)
    {}
};
```

セクション 3.2 の ApplyFoo クラスとは異なることに注意してください。まず、operator() は const *ではありません*。これは、SumFoo::sum を更新しなければならないためです。2 番目に、SumFoo には、*分割コンストラクター* があり、また parallel_reduce が機能するために join メソッドがなければなりません。分割コンストラクターは、引数として元のオブジェクトへの参照と、ライブラリーにより定義される仮引数の型 split をとります。この仮引数によって、分割コンストラクターとコピー・コンストラクターが区別されます。

タスク・スケジューラーで判断され、ワーカースレッドが利用できる場合は、プロセッサーのサブタスクを作成する分割コンストラクターを起動することにより、parallel_reduce がそのワーカースレッドに作業を受け渡します。タスクが終了すると、parallel_reduce は join メソッドを使用して、サブタスク結果を累積します。次のグラフは、分割と結合のシーケンスを示しています。

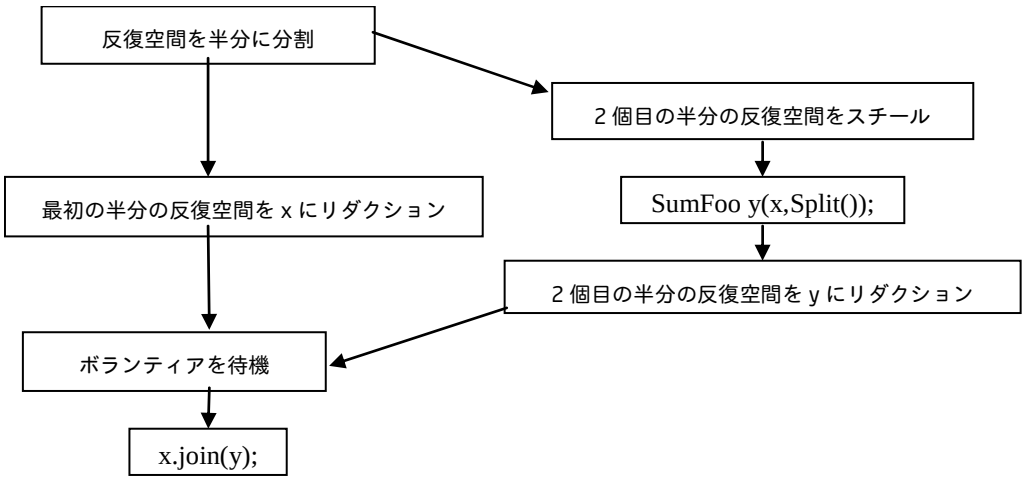


図 3: 分割と結合のシーケンス

図 3 の矢印は、時間順を示します。最初の半分のリダクションにオブジェクト x が使用される間に、分割コンストラクターが同時に実行される可能性があることに注意してください。そのため、y を作成する分割コンストラクターのすべての動作が、x に関してスレッドセーフである必要があります。分割コンストラクターでほかのオブジェクトと共有の参照カウントをインクリメントする必要がある場合は、アトミック・インクリメントを使用しなければなりません。

```
float ParallelSumFoo( const float a[], size_t n ) {
    SumFoo sf(a);
    parallel_reduce(blocked_range<size_t>(0,n,IdealGrainSize), sf );
    return sf.sum;
}
```

parallel_for の場合、少なくとも 10,000 命令を実行できるだけの十分な反復回数を確保した妥当な grainsize を指定する必要があります。妥当な値がわからない場合は、grainsize を多めに指定してください。また、パーティショナー・オブジェクトを使用して、ランタイム・ライブラリーにより範囲をチャンクに分割する指針とすることもできます (3.2.3)。

parallel_reduce は、あらゆる結合操作を一般化します。一般に、分割コンストラクターでは、2 つのことが行われます。

- ループ本体の実行に必要な読み取り専用情報をコピーする。
- リダクション変数を初期化して、操作要素を特定する。

join メソッドは、対応するマージ操作を行います。複数のリダクションを同時に行うことが可能です。単一の parallel_reduce で min と max を求めることができます。

3.3.1 サンプル (上級編)

より高度な結合操作の例として、Foo(i) が最小化されたインデックスを求めます。直列バージョンは次のとおりです。

```
long SerialMinIndexFoo( const float a[], size_t n ) {
    float value_of_min = FLT_MAX;       // FLT_MAX from <climits>
    long index_of_min = -1;
    for( size_t i=0; i<n; ++i ) {
        float value = Foo(a[i]);
        if( value<value_of_min ) {
                value_of_min = value;
            index_of_min = i;
        }
    }
    return index_of_min;
}
```

これまでに検出した最小値と値のインデックスをトラッキングすることでループは動作します。これは、ループの反復間で運搬される唯一の情報です。parallel_reduce を使用してループを変換するには、関数オブジェクトは運搬情報と反復が複数のスレッドに分散された場合に、この情報をマージする方法を把握しなければなりません。また、関数オブジェクトは、コンテキストを提供するため、a へのポインターを記録しなければなりません。

次のコードは、完全な関数オブジェクトです。

```
class MinIndexFoo {
    const float *const my_a;
public:
    float value_of_min;
```

```
    long index_of_min;
    void operator()( const blocked_range<size_t>& r ) {
        const float *a = my_a;
        for( size_t i=r.begin(); i!=r.end(); ++i ) {
           float value = Foo(a[i]);
           if( value<value_of_min ) {
               value_of_min = value;
               index_of_min = i;
           }
        }
    }

    MinIndexFoo( MinIndexFoo& x, split ) :
        my_a(x.my_a),
        value_of_min(FLT_MAX),    // FLT_MAX from <climits>
        index_of_min(-1)
   {}

    void join( const SumFoo& y ) {
        if( y.value_of_min<x.value_of_min ) {
            value_of_min = y.value_of_min;
            index_of_min = y.index_of_min;
        }
    }

    MinIndexFoo( const float a[] ) :
        my_a(a),
        value_of_min(FLT_MAX),    // FLT_MAX from <climits>
        index_of_min(-1),
    {}
};
```

SerialMinIndex は、次のように parallel_reduce を使用して、記述を変更することができます。

```
long ParallelMinIndexFoo( float a[], size_t n ) {
    MinIndexFoo mif(a);
    parallel_reduce(blocked_range<size_t>(0,n,IdealGrainSize), mif );
    return mif.index_of_min;
}
```

examples/parallel_reduce/primes ディレクトリーには、parallel_reduce に基づく素数検索プログラムが含まれています。

3.4 トピック (上級編): 異なる種類の反復空間

これまでの例では、範囲の指定に blocked_range<T> クラスを使用しました。このクラスは、多くの状況で役立ちますが、すべての状況で適切であるとは限りません。インテル® スレッディング・ビルディング・ブロックを使用すると、独自の反復空間オブジェクトを定義できます。オブジェクトは、2

つのメソッドと “分割コンストラクター” を使用して、サブ空間にオブジェクトを分割する方法を指定する必要があります。クラスが R により呼び出された場合、メソッドとコンストラクターは次のようになります。

```
class R {
    // True if range is empty
    bool empty() const;
    // True if range can be split into non-empty subranges
    bool is_divisible() const;
    // Split r into subranges r and *this
    R( R& r, split );
    ...
};
```

empty メソッドは、範囲が空の場合に true を返します。is_divisible メソッドは、範囲が 2 つの空ではないサブ範囲に分割された場合に true を返します。そのような分割はオーバーヘッドが生じても有用です。分割コンストラクターの引数は 2 つあります。

- 最初は型 R
- 2 番目は型 tbb::split

2 番目の引数は使用されません。この引数は、コンストラクターを通常のコピー・コンストラクターと区別するために用いられます。分割コンストラクターは、r を 2 つに分割し、r を最初の半分として更新し、構築されたオブジェクトを残りの半分とします。これら 2 つは空ではありません。並列アルゴリズムのテンプレートは、r.is_divisible が true の場合のみ、分割コンストラクターを r 上で呼び出します。

反復空間は、線形である必要はありません。2 次元の範囲例、tbb/blocked_range2d.h を参照してください。分割コンストラクターは、範囲を最長の軸に沿って分割しようとします。parallel_for とともに使用すると、キャッシュの使用を向上させる方法で、ループは “再帰的にブロック” されます。このキャッシュの動作は、parallel_for を blocked_range2d<T> に対して使用することで、単一プロセッサーでも、同等のシーケンシャルな場合よりループの実行が速くなることを意味します。

3.4.1 コードサンプル

examples/parallel_for/seismic ディレクトリーには、parallel_for と blocked_range に基づいた地震波シミュレーションが含まれています。また、examples/parallel_for/tacheon ディレクトリーには、parallel_for と blocked_range2d に基づいた、より複雑なレイ・トレーシングのサンプルが含まれています。

4 複雑なループの並列化

3 章では、構造体のみを使用して多くのアプリケーションの並列化を行いました。しかし、状況によっては、ほかの並列化パターンも必要になります。このセクションでは、これらの代替パターンのサポートについて説明します。

4.1 完了までの並列化: parallel_while

いくつかのループでは、反復空間の最後が前もって知らされていないか、あるいはループ本体が、ループが終了する前に反復を追加することがあります。どちらの場合も、tbb::parallel_while テンプレート・クラスを使用することで対処できます。

リンクリストは、前もって知らされていない反復空間の例です。並列プログラミングでは、リンクリストの項目へのアクセスは本質的に直列であるため、リンクリストの代わりに動的配列を使用するほうが効率的です。しかし、リンクリストを使用する必要があり、項目が安全に並列処理可能で、各項目の処理に少なくとも数千命令がかかる場合、直列形式が次のようになる状況では、parallel_while を使用することができます。

```
void SerialApplyFooToList( Item*root ) {
    for( Item* ptr=root; ptr!=NULL; ptr=ptr->next )
        Foo(pointer->data);
}
```

Foo の実行に少なくとも数千命令がかかる場合、parallel_while を使用するようにループを変換することで、並列処理を高速化できます。以前に説明したテンプレートとは異なり、parallel_while は関数ではなくクラスで、*2 つの*ユーザー定義オブジェクトが必要です。最初のオブジェクトは、項目のストリームを定義します。このオブジェクトには、pop_if_present メソッドが含まれます。bool b = pop_if_present(v) が呼び出されたとき、次の反復がある場合は、v を次の反復値に設定して true を返します。次の反復がない場合は、false を返します。青のフォントは、インテル® スレッディング・ビルディング・ブロックで必要なコードの変更を示します。黒のフォントは、オリジナルコードの一部で、ストリーム形式に変換するために変更されたコードを示します。

```
class ItemStream {
    Item* my_ptr;
public:
    bool pop_if_present( Item*& item ) {
        if( my_ptr ) {
```

```
            item = my_ptr;
            my_ptr = my_ptr->next;
            return true;
        } else {
            return false;
        }
    };
    ItemStream( Item* root ) : my_ptr(root) {}
}
```

2 番目のオブジェクトは、ループ本体を定義します。const operator() およびメンバー型 argument_type を定義する必要があります。これは、const でなければならない点を除いて、C++ 標準ヘッダー <functional> の C++ 関数オブジェクトと似ています。

```
class ApplyFoo {
public:
    void operator()( Item* item ) const {
        Foo(item->data);
    }
    typedef Item* argument_type;
};
```

ストリームとボディークラスを指定します。コードは次のようになります。

```
void ParallelApplyFooToList( Item*root ) {
    parallel_while<ApplyFoo> w;
    ItemStream stream;
    ApplyFoo body;
    w.run( stream, body );
}
```

parallel_while は、同じストリームで同時に pop_if_present メソッドを呼び出さないため、このメソッドは指定されたストリームでスレッドセーフである必要はありません。フェッチが直列化されるため、parallel_while はスケーラブルでなくなる点に注意してください。しかし、多くの状況では、直列に処理した場合よりも高速化されます。

次に、parallel_while をスケーラブルにできる別の方法を説明します。parallel_while のボディー *w* (構築されたときに *w* への参照が指定された場合) は、w.add(*item*) を呼び出してほかの作業を追加することができます。ここで、*item* は Body::argument_type 型です。例えば、ツリーのノードを処理することは、その子孫を処理するための前提条件です。parallel_while では、ノードを処理した後、parallel_while::add を使用して子孫ノードを追加することができます。parallel_while のインスタンスは、すべての項目が処理されるまで終了しません。

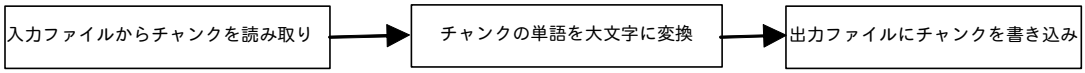
4.1.1 コードサンプル

examples/parallel_while/parallel_preorder ディレクトリーには、parallel_while を使用して、非巡回有向グラフの並列の前順走査を行う小さなアプリケーションが含まれています。

4.2 組み立てラインでの作業: パイプライン

パイプライン は、従来の生産組み立てラインを模倣した一般的な並列パターンです。データは一連のパイプライン・ステージを流れ、各ステージではいくつかの方法でデータが処理されます。データの入力ストリームが与えられると、これらのステージのいくつかは並列で操作することができます。ほかのステージは並列で操作することはできません。例えば、ビデオ処理では、フレーム上のいくつかの操作はほかのフレームに依存しないため、複数のフレーム上で同時に操作することができます。一方、前のフレームを最初に処理しなければならない操作もあります。

インテル® スレッディング・ビルディング・ブロックの pipeline クラスと filter クラスは、パイプライン・パターンを実装します。単純なテキスト処理問題を使用して、pipeline と filter の使用方法について説明します。テキストファイルを読み取り、各単語の先頭の文字を大文字にして、変更したテキストを新しいファイルに書き込む問題です。次にパイプラインの図を示します。



ファイル I/O は順次であると仮定します。しかし、先頭の文字を大文字にするステージは並列に行うことができます。つまり、*n* チャンクを非常に速く直列に読み取ることができれば、それらが適切な順で出力ファイルに書き込まれている限り、*n* チャンクをそれぞれ並列に大文字にすることができます。

文字を大文字にするべきかどうかを決定するために、*前の* 文字が空白かどうかを調べます。各チャンクの最初の文字について、前のチャンクの最後の文字を調べます。しかし、そうすると、中間ステージでの依存関係が複雑になります。解決方法は、各チャンクと前のチャンクの最後の文字を格納することです。チャンクは、1 文字ずつオーバーラップします。この "オーバーラップ・ウィンドウ" 手法は、パイプライン処理問題で非常に一般的です。例では、ウィンドウは MyBuffer クラスのインスタンスで表されています。この例は、begin()[-1] が正当で前のチャンクの最後の文字を保持することを除けば、文字用の典型的な STL コンテナーのように見えます。

```
// Buffer that holds block of characters and last character of previous
buffer.
class MyBuffer {
    static const size_t buffer_size = 10000;
    char* my_end;
```

```
    // storage[0] holds the last character of the previous buffer.
    char storage[1+buffer_size];
public:
    // Pointer to first character in the buffer
    char* begin() {return storage+1;}
    const char* begin() const {return storage+1;}
    // Pointer to one past last character in the buffer
    char* end() const {return my_end;}
    // Set end of buffer.
    void set_end( char* new_ptr ) {my_end=new_ptr;}
    // Number of bytes a buffer can hold
    size_t max_size() const {return buffer_size;}
    // Number of bytes in buffer.
    size_t size() const {return my_end-begin();}
};
```

次は、パイプラインの構築および実行用のトップレベル・コードです

```
    // Create the pipeline
    tbb::pipeline pipeline;

    // Create file-reading writing stage and add it to the pipeline
    MyInputFilter input_filter( input_file );
    pipeline.add_filter( input_filter );

    // Create capitalization stage and add it to the pipeline
    MyTransformFilter transform_filter;
    pipeline.add_filter( transform_filter );

    // Create file-writing stage and add it to the pipeline
    MyOutputFilter output_filter( output_file );
    pipeline.add_filter( output_filter );

    // Run the pipeline
    pipeline.run( MyInputFilter::n_buffer );

    // Remove filters from pipeline before they are implicitly destroyed.
    pipeline.clear();
```

pipeline::run メソッドのパラメーターは、並列処理のレベルを制御します。概念的には、トークンはパイプラインを流れます。直列ステージでは、各トークンは直列で順番に処理されます。並列ステージでは、複数のトークンを並列に処理することができます。トークンの数に制限がない場合、出力ステージの処理が追いつかないため、トークンを取得する中間ステージで問題が発生することがあります。通常、この状況では、中間ステージで不適切にリソースが消費されます。pipeline::run メソッドのパラメーターは、処理できるトークンの最大数を指定します。いったんこの制限に達すると、別のトークンが出力ステージで破棄されるまで、pipeline クラスは入力ステージで新しいトークンを作成しません。

このトップレベルのコードには、パイプラインからすべてのステージを削除する clear メソッドも含まれています。パイプラインの前にフィルターを破棄しなければならない場合、この呼び出しが必要です。パイプラインはフィルターを保持するコンテナーです。C++ のほとんどのコンテナーと同様に、項目がコンテナーにある間に破棄することは不正です。

では、ステージがどのように定義されるか、詳細に見ていきましょう。各ステージは、filter クラスから派生します。最初に、最も簡単な出力ステージについて考えます。

```
// Filter that writes each buffer to a file.
class MyOutputFilter: public tbb::filter {
    FILE* my_output_file;
public:
    MyOutputFilter( FILE* output_file );
    /*override*/void* operator()( void* item );
};

MyOutputFilter::MyOutputFilter( FILE* output_file ) :
    tbb::filter(/*is_serial=*/true),
    my_output_file(output_file)
{
}

void* MyOutputFilter::operator()( void* item ) {
    MyBuffer& b = *static_cast<MyBuffer*>(item);
    fwrite( b.begin(), 1, b.size(), my_output_file );
    return NULL;
}
```

パイプライン・テンプレートに"フックする"部分は、青で示されています。クラスは、filter クラスから派生します。そのコンストラクターが filter の基本クラス・コンストラクターを呼び出す場合、これが直列フィルターであることを指定します。クラスは、項目を処理するためにパイプラインによって呼び出される仮想メソッド filter::operator() をオーバーライドします。item パラメーターは、処理される項目を指します。返される値は、次のフィルターで処理される項目を指します。これは最後のフィルターなので、返される値は無視され、NULL になります。

中間ステージも同様です。operator() は、次のステージに送られる項目へのポインターを返します。

```
// Filter that changes the first letter of each word
// from lower case to upper case.
class MyTransformFilter: public tbb::filter {
public:
    MyTransformFilter();
    /*override*/void* operator()( void* item );
};

MyTransformFilter::MyTransformFilter() :
    tbb::filter(/*serial=*/false)
{}
```

```
/*override*/void* MyTransformFilter::operator()( void* item ) {
    MyBuffer& b = *static_cast<MyBuffer*>(item);
    bool prev_char_is_space = b.begin()[-1]==' ';
    for( char* s=b.begin(); s!=b.end(); ++s ) {
        if( prev_char_is_space && islower(*s) )
            *s = toupper(*s);
        prev_char_is_space = isspace(*s);
    }
    return &b;
}
```

また、このステージは、純粋なローカルデータ上で動作します。したがって、`operator()` 上の任意の数の呼び出しは、`MyTransformFilter` の同じインスタンス上で同時に実行することができます。この事実は、`<serial>` パラメーターを false に設定して基本クラス `filter` を構築することでパイプラインに伝えられます。

いつ入力の最後に達したかを判断してバッファーを割り当てなければならないため、入力フィルターは最も複雑です。

```
class MyInputFilter: public tbb::filter {
public:
    static const size_t n_buffer = 4;
    MyInputFilter( FILE* input_file_ );
private:
    FILE* input_file;
    size_t next_buffer;
    char last_char_of_previous_buffer;
    MyBuffer buffer[n_buffer];
    /*override*/ void* operator()(void*);
};

MyInputFilter::MyInputFilter( FILE* input_file_ ) :
    filter(/*is_serial=*/true),
    next_buffer(0),
    input_file(input_file_),
    last_char_of_previous_buffer(' ')
{
}

void* MyInputFilter::operator()(void*) {
    MyBuffer& b = buffer[next_buffer];
    next_buffer = (next_buffer+1) % n_buffer;
    size_t n = fread( b.begin(), 1, b.max_size(), input_file );
    if( !n ) {
             // end of file
        return NULL;
    } else {
        b.begin()[-1] = last_char_of_previous_buffer;
        last_char_of_previous_buffer = b.begin()[n-1];
        b.set_end( b.begin()+n );
        return &b;
```

```
    }
}
```

入力フィルターは順次ファイルから読み取りを行っているので直列です。後のバージョンでは変更されるかもしれませんが、現在の実装では、`is_serial` パラメーターは最初のステージでは無視されます。`operator()` のオーバーライドは、変換でなく、ストリームを生成するため、そのパラメーターを無視します。前のチャンクの最後の文字を保持しているため、適切にウィンドウをオーバーラップすることができます。

バッファーは、`n_buffer` サイズの循環キューから割り当てられます。最初の `n_buffer` 入力操作の後、バッファーが使用中かどうかが確認されることなく再利用されるため、この処理は危険に思えますが、次の 2 つの制約があるため、再利用は安全です。

- パイプラインは、`pipeline::run` が呼び出されたときに `n_buffer` トークンを受け取る。したがって、`n_buffer` よりも多くのバッファーが同時に処理されることはありません。
- 最後のステージは直列である。したがって、バッファーは最初のステージで割り当てられたために最後のステージでリタイアされます。

最後のステージが直列 *でない* 場合、バッファーはアウトオブオーダーでリタイアするため、どのバッファーが現在使用中であるかを追跡する必要があります。

`examples/pipeline/text_filter` ディレクトリーには、テキストフィルター用の完全なコードが含まれています。

4.2.1 パイプラインの処理能力

`pipeline` の処理能力は、トークンが流れる比率における 2 つの条件によって制限されます。最初に、`pipeline` が *N* 個のトークンで実行される場合、*N* よりも多くの操作を並列に実行できないことは明白です。*N* の正しい値を選択するには試行が必要です。値が非常に低いと、並列処理が制限されます。値が非常に高いと、より多くのリソース (例えば、より多くのバッファー) が必要になります。次に、パイプラインの処理能力は最も遅い直列ステージの処理能力に制限されます。この制約は並列ステージのないパイプラインにも当てはまります。ほかのステージがどれだけ速くても、最も遅い直列ステージがボトルネックになります。したがって、一般に、直列ステージを速い状態で保つ必要があります。可能であれば、作業を並列ステージに移すようにしてください。

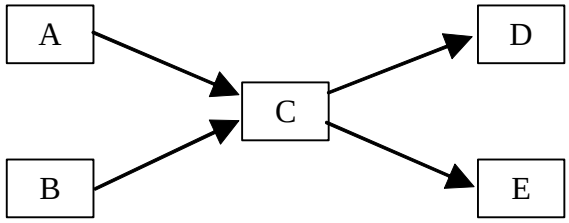
直列ステージがシステムの I/O 速度によって制限されているため、テキスト処理の例はあまり速くなりません。ファイルがローカルディスク上にある場合でも、2 倍以上のスピードアップは見込めません。

実際に pipeline の恩恵を得るには、並列ステージが直列ステージよりも重い処理を行う必要があります。

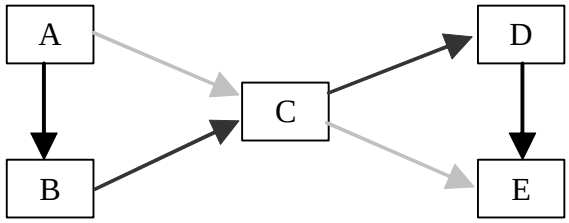
各トークンのウィンドウサイズ、またはサブ問題サイズも処理能力を制限します。ウィンドウサイズを非常に小さくすると、オーバーヘッドが大きくなります。ウィンドウサイズを非常に大きくすると、キャッシュに収まらなくなります。キャッシュに収まる、できるだけ大きなウィンドウサイズが理想的です。最適なウィンドウサイズを見つけるには、多少の試行が必要です。

4.2.2 非線形パイプライン

pipeline テンプレートは、線形パイプラインのみをサポートします。次のダイアグラムのようなブロック様式の配管工事 (柔軟性のない構成) を直接処理することはありません。



しかし、pipeline を使用することはできます。次のように、ステージを線形順に位相的にソートするだけです。



薄い灰色の矢印は、ほかの矢印の遷移的な閉包によって暗に示されるオリジナルの矢印です。並列処理の多くがステージを線形順にすることで失われているように見えますが、実際には、唯一の損失は処理能力ではなくパイプラインの*レイテンシー*によるものです。レイテンシーは、トークンがパイプラインの最初から最後までを流れる時間です。十分な数のプロセッサーが提供されると、オリジナルの非線形パイプラインのレイテンシーは 3 ステージになります。これは、ステージ A と B およびステージ D と E が同時にトークンを処理することができるためです。線形パイプラインでは、レイテンシーは 5 ステージになります。上記のステージ A、B、D および E の動作は、パイプラインの次のステージに渡すこと以外に、ステージで動作する必要のないオブジェクトを適切に制御するように修正する必要があります。

位相に関係なく、処理能力は最も遅い直列ステージによってまだ制限されているため、処理能力は同じままです。`pipeline` が非線形パイプラインをサポートした場合、プログラミングは複雑になりますが、処理能力はそれほど向上しません。`pipeline` の直列制限は、パフォーマンスとプログラミングの労力のバランスをうまく取っています。

4.3 ループのまとめ

インテル® スレッディング・ビルディング・ブロックのハイレベルなループ・テンプレートは、ゼロから開始することなく、マルチコアチップの能力を活かす効率的でスケーラブルな方法を提供します。ソフトウェアを高いタスク・パターン・レベルで設計でき、スレッドの低レベルな操作について考慮する必要がありません。ループ・テンプレートは汎用的なので、特定の用途に応じてカスタマイズすることができます。ループ・テンプレートを使用して、マルチコアの能力を最大限に活用してください。

5 コンテナー

インテル® スレッディング・ビルディング・ブロックは、高度なコンカレント・コンテナー・クラスを提供します。これらのコンテナーは、ロー Windows / Linux スレッド、またはタスクベースのプログラミング (10.1) とともに使用することができます

コンカレント・コンテナーは、複数のスレッドが同時にコンテナーの項目にアクセスして更新できるようにします。典型的な C++ STL コンテナーでは同時更新は許可されません。同時に更新すると、コンテナーが破損する恐れがあります。STL コンテナーは、同時アクセスで安全になるように mutex でラップして、コンテナー上で一度に 1 つのスレッドのみを操作するようにします。しかし、このアプローチでは並列化が行われないため、並列処理によるスピードアップは制限されます。

インテル® スレッディング・ビルディング・ブロックに含まれているコンテナーは、次の 1 つまたは両方のメソッドにより、非常に高いレベルの並列化を提供します。

- **ファイングレイン・ロッキング**。ファイングレイン・ロッキングでは、複数のスレッドは実際にロックする必要のある部分だけをロックしてコンテナー上で動作します。異なるスレッドが異なる部分にアクセスする限り、同時に処理することができます。
- **ロックフリー・アルゴリズム**。ロックフリー・アルゴリズムでは、異なるスレッドはほかの干渉するスレッドの影響を計算して修正します。

高度なコンカレント・コンテナーの使用にはコストがかかる点に注意してください。通常、一般的な STL コンテナーよりもオーバーヘッドが大きくなります。高度なコンカレント・コンテナーでの操作は、STL コンテナーでの操作よりも時間がかかります。したがって、高度なコンカレント・コンテナーによって可能になる追加の並列化が、より遅い順次パフォーマンスよりも重要である場合に、高度なコンカレント・コンテナーを使用してください。

5.1 concurrent_hash_map

`concurrent_hash_map<`*Key*, *T*, *HashCompare*`>` は、同時アクセスを許可するハッシュテーブルです。テーブルは、キーから *T* 型へのマップです。`HashCompare` 型は、キーをハッシュする方法と 2 つのキーを比較する方法を定義します。

次の例は、キーが文字列で、対応するデータは配列 Data で各文字列が現れる回数である、concurrent_hash_map を構築します。

```
#include "tbb/concurrent_hash_map.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
#include <string>

using namespace tbb;
using namespace std;

// Structure that defines hashing and comparison operations for user's type.
struct MyHashCompare {
    static size_t hash( const string& x ) {
        size_t h = 0;
        for( const char* s = x.c_str(); *s; ++s )
            h = (h*17)^*s;
        return h;
    }
    //! True if strings are equal
    static bool equal( const string& x, const string& y ) {
        return x==y;
    }
};

// A concurrent hash table that maps strings to ints.
typedef concurrent_hash_map<string,int,MyHashCompare> StringTable;

// Function object for counting occurrences of strings.
struct Tally {
    StringTable& table;
    Tally( StringTable& table_ ) : table(table_) {}
    void operator()( const blocked_range<string*> range ) const {
        for( string* p=range.begin(); p!=range.end(); ++p ) {
            StringTable::accessor a;
            table.insert( a, *p );
            a->second += 1;
        }
    }
};

const size_t N = 1000000;

string Data[N];

void CountOccurrences() {
    // Construct empty table.
    StringTable table;

    // Put occurrences into the table
         parallel_for( blocked_range<string*>( Data, Data+N, 100 ),
Tally(table) );

    // Display the occurrences
```

```
    for( StringTable::iterator i=table.begin(); i!=table.end(); ++i )
        printf("%s %d\n",i->first.c_str(),i->second);
}
```

concurrent_hash_map は、std::pair<const *Key,T*> 型の要素のコンテナーとして動作します。通常、コンテナー要素にアクセスする場合、要素の更新または読み取りを行います。concurrent_hash_map テンプレート・クラスは、スマートポインターとして動作する accessor クラスと const_accessor クラスを使用して 2 つの目的をそれぞれサポートします。*accessor* は、*update* (*write*) アクセスを表します。要素を指す限り、accessor が完了するまで、テーブルのキーを調べる操作はすべてブロックされます。const_accessor は、*read-only* アクセスを指すことを除いて同じです。複数の const_accessors が同時に同じ要素を指すことができます。この機能は、要素が頻繁に読み取りされ、ほとんど更新されない状況の並列化を大幅に向上させます。

find メソッドおよび insert は、accessor または const_accessor を引数として使用します。この選択は、concurrent_hash_map に *update* と *read-only* アクセスのどちらを要求しているかを知らせます。いったんメソッドが返されると、accessor または const_accessor が破棄されるまでアクセスは続きます。要素へのアクセスはほかのスレッドをブロックするため、accessor または const_accessor の生存期間を短くしてください。このためには、最も内側のブロックで宣言を行ってください。アクセスをブロックの終了よりも早くリリースするには、release メソッドを使用してください。次の例は、破棄に依存してスレッドの生存期間を終了する代わりに、release を使用するループ本体の再処理です。

```
        StringTable accessor a;
        for( string* p=range.begin(); p!=range.end(); ++p ) {
            table.insert( a, *p );
            a->second += 1;
            a.release();
        }
```

remove(key) メソッドも同時に操作することができます。このメソッドは書き込みアクセスを暗黙的に要求します。したがって、キーを削除する前に、key でほかに残っているアクセスを待機します。

5.1.1 HashCompare の詳細

一般的に、HashCompare の定義は 2 つの署名を提供します。

- *Key* を size_t にマップする hash メソッド
- 2 つのキーが等しいかどうかを判断する equal メソッド

*2 つのキーが等しい場合、同じ値にハッシュしなければならない*ため、署名は単一クラスでともに移動します。そうしないと、ハッシュテーブルは動作しません。常に `0` にハッシュすれば当然この要件を満たすことができますが、非常に効率が悪くなります。各キーが異なる値にハッシュされるのが理想的です。少なくとも、2 つの別のキーが同じ値にハッシュされる可能性を低くしておく必要があります。

異なるインスタンスで異なる動作が必要でなければ、*HashCompare* のメソッドは `static` です。この場合、*HashCompare* をパラメーターとするコンストラクターを使用して `concurrent_hash_map` を構築します。次の例は、インスタンス依存のメソッドを使用した以前の例のバリエーションです。インスタンスは、内部フラグ `ignore_case` に基づいて、大文字/小文字を区別するハッシュまたは区別しないハッシュを行い、比較します。

```
// Structure that defines hashing and comparison operations
class VariantHashCompare {
    // If true, then case of letters is ignored.
    bool ignore_case;
public:
    size_t hash( const string& x ) {
        size_t h = 0;
        for( const char* s = x.c_str(); *s; s++ )
            h = (h*17)^*(ignore_case?tolower(*s):*s);
        return h;
    }
    // True if strings are equal
    bool equal( const string& x, const string& y ) {
        if( ignore_case )
            strcasecmp( x.c_str(), y.c_str() )==0;
        else
            return x==y;
    }
    VariantHashCompare( bool ignore_case_ ) : ignore_case() {}
};

typedef concurrent_hash_map<string,int, VariantHashCompare>
VariantStringTable;

VariantStringTable CaseSensitiveTable(VariantHashCompare(false));
VariantStringTable CaseInsensitiveTable(VariantHashCompare(true));
```

`examples/concurrent_hash_map/count_strings` ディレクトリーには、`concurrent_hash_map` を使用して複数のプロセッサーがヒストグラムを協調して構築できるようにする完全な例が含まれています。

5.2 concurrent_vector

concurrent_vector<*T*> は、*T* の動的に拡張可能な配列です。ほかのスレッドをその要素上で操作しているか、自身が拡張しているときに concurrent_vector を拡張させることは安全です。安全な同時拡張のため、concurrent_vector には、動的配列の共通使用をサポートするサイズ変更として、grow_by と grow_to_at_least の 2 つのメソッドがあります。

grow_by(n) メソッドは、*n* 個の連続する要素をベクトルに安全に追加して、最初に追加した要素のインデックスを返します。各要素は *T*() で初期化されます。例えば、次のルーチンは C 文字列を共有ベクトルに安全に追加します。

```
void Append( concurrent_vector<char>& vector, const char* string ) {
    size_t n = strlen(string)+1;
    memcpy( &vector[vector.grow_by(n)], string, n+1 );
}
```

関連する grow_to_at_least(*n*) メソッドは、ベクトルがサイズ *n* よりも短い場合、ベクトルをそのサイズまで拡張させます。grow_by および grow_to_at_least への同時呼び出しは、ベクトルに追加された順に要素を返す必要はありません。

size() メソッドは、grow_by メソッドと grow_to_at_least メソッドによってまだ同時構築されている要素を含む、メソッドベクトルの要素の数を返します。また、イテレーターが end() の現在の値を超えて移動しない限り、concurrent_vector が拡張する間、イテレーターを使用することは安全です。しかし、イテレーターは同時構築されている要素を参照することがあります。そのため、構築とアクセスを同期させる必要があります。

concurrent_vector<*T*> は、配列が消去されるまで要素を移動しません。これは、シングルスレッド・コードであっても STL std::vector より利点があります。しかし、concurrent_vector は std::vector よりもオーバーヘッドが大きくなります。ほかのアクセスの処理中に動的にサイズ変更する機能が実際に必要な場合、または要素を移動してはならない場合のみ、concurrent_vector を使用してください。

5.2.1 消去は並列化セーフではない

注意: concurrent_vector の操作は、ベクトルの消去や破棄ではなく、*拡張* に関して並列化セーフです。concurrent_vector で処理を行っているほかの操作がある場合、clear() メソッドを呼び出さないでください。

5.3 concurrent_queue

concurrent_queue<*T*> テンプレート・クラスは、型 T の値の同時キューを実装します。複数のスレッドはキューから要素を同時にプッシュおよびポップします。

同時キューの動作には注意してください。シングルスレッド・プログラムでは、キューは FIFO (先入れ先出し) 構造です。しかし、マルチスレッド・プログラムの場合、プッシュとポップが同時に行われるため、"先" の定義は不確定です。concurrent_queue は、スレッドが 2 つの値をプッシュして、別のスレッドがその 2 つの値をポップする場合に、プッシュされたのと同じ順で値がポップされることを保証します。

ポップには、ブロックと非ブロックがあります。pop_if_present メソッドは、非ブロックです。値をポップしようとして、キューが空のためにポップできない場合、いずれにしてもリターンします。pop メソッドは、値をポップするまでブロックします。項目が利用可能になるまでスレッドが待たなければならず、ほかに何も行っていない場合、while(!pop_if_present(item)) continue; ではなく、pop(item) を使用してください。pop はプロセッサー・リソースをループより効率的に使用します。

大部分の STL コンテナーとは異なり、concurrent_queue::size_type は符号なしではなく、*符号付き* 整数型です。これは、concurrent_queue::size() が開始したプッシュ操作の数から、開始したポップ操作の数を引いた値として定義されるためです。ポップの数がプッシュの数より多い場合、size() は負になります。例えば、concurrent_queue が空で保留中のポップ操作が *n* ある場合、size() は –*n* を返します。この方法を使用すると、生産者は、キュー上で待機している消費者の数を簡単に知ることができます。empty() メソッドは、size() が負でない場合にのみ true として定義されます。

デフォルトでは、concurrent_queue<T> は無制限です。メモリーがなくなるまで、任意の数の値を保持します。set_capacity メソッドでキューのキャパシティーを設定して制限することができます。キャパシティーを設定すると、キューに空間ができるまでプッシュはブロックされます。制限のあるキューは無制限のキューよりも遅いため、キューが大きくなりすぎることを防ぐコードがプログラム中にある場合は、キャパシティーを設定しないほうが良いでしょう。

5.3.1 デバッグ目的の concurrent_queue 上での反復

concurrent_queue テンプレート・クラス は、STL スタイルの反復をサポートします。キューをダンプする必要がある場合、このサポートはデバッグ目的でのみ行われます。イテレーターは前方へのみ移動します。非常に遅いため、製品コードでは使用しないでください。キューが変更された場合、キューを指すイテレーターはすべて無効になり、安全に使用できなくなります。次のコードは、キューをダンプします。operator<< は、Foo 用に定義されます。

```
concurrent_queue<Foo> q;
...
for(concurrent_queue<Foo>::const_iterator i(q.begin()); i!=q.end(); ++i ) {
    cout << *i;
}
```

5.3.2 キューを使用するべきではない状況

キューは、生産者から消費者をバッファーするため並列プログラムで広く使用されています。しかし、明示的なキューを使用する前に、代わりに parallel_while (4.1) または pipeline (4.2) を使用することを検討してください。これらの操作は、多くの場合、次の理由によりキューより効率的です。

- FIFO (先入れ先出し) 順を維持しなければならないため、キューは本質的なボトルネックである。
- 値をポップしているスレッドは、値がプッシュされるまでアイドル状態で待たなければならない。
- キューは受動的なデータ構造である。スレッドが値をプッシュした場合、値をポップするまで時間がかかり、値 (およびその参照) はキャッシュにない "コールド" 状態になります。また、さらに悪い状態では、別のスレッドが値をポップし、値 (およびその参照) を別のプロセッサーに移動しなければなりません。

対照的に、parallel_while および pipeline はこれらのボトルネックを回避します。これらのスレッド化は暗黙的であるため、値が現れるまでほかの作業を行うようにワーカースレッドを最適化します。また、キャッシュ上の項目をホットな状態で維持しようとします。例えば、別の作業項目が parallel_while に追加された場合、"ホット" スレッドが項目を処理する前に別の使用されていないスレッドが項目をスチールすることができなければ、項目を追加したスレッドに対して局所的なままです。このように、項目はホットスレッドでより多く処理されます。

5.4 コンテナーのまとめ

インテル® スレッディング・ビルディング・ブロックのハイレベルなコンテナーは、同時アクセスで一般的な慣用句を有効にします。これらは、ラッパーがまわりをロックする直列コンテナーになる状況に適しています。

6 排他制御 (mutex)

排他制御は、コード領域を同時に実行できるスレッドの数を制御します。インテル® スレッディング・ビルディング・ブロックでは、排他制御は *mutex* と *lock* によって実装されます。mutex は、スレッドがロックを取得できるオブジェクトです。一度に 1 つのスレッドのみ mutex でロックを取得することができます。ほかのスレッドは、順番になるまで待たなければなりません。

最も簡単な mutex は、`spin_mutex` です。`spin_mutex` でロックを取得しようとするスレッドは、ロックを取得できるようになるまで、ビジーウェイトを行います。`spin_mutex` は、ロックが少数の命令に適用されている場合に適しています。例えば、次のコードは、mutex `FreeListMutex` を使用して共有変数 FreeList を保護します。一度に単一のスレッドだけが FreeList にアクセスするようにチェックします。黒のフォントは、通常の順次コードを示します。青のテキストは、コードをスレッドセーフにするために追加されたコードを示します。

```
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;

Node* AllocateNode() {
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}

void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
```

`scoped_lock` のコンストラクターは、`FreeListMutex` でほかのロックがなくなるまで待機します。デストラクターはロックをリリースします。`AllocateNode` ルーチン内の中括弧内のコードは、通常とは異なり、その役割は、ほかのスレッドができるだけ早くロックが得られるように、ロック期間を短くするようにしています。

注意: 必ずロック・オブジェクトを指定してください。指定されない場合は、すぐに破棄されます。例えば、例の scoped_lock オブジェクトの作成を

```
FreeListMutexType::scoped_lock (FreeListMutex);
```

に変更すると、scoped_lock は実行がセミコロンに達したときに破棄され、FreeList がアクセスされる*前に* ロックをリリースします。

AllocateNode の別の記述方法は次のとおりです。

```
Node* AllocateNode() {
    Node* n;
    FreeListMutexType::scoped_lock lock;
    lock.acquire(FreeListMutex);
    n = FreeList;
    if( n )
        FreeList = n->next;
    lock.release();
    if( !n )
        n = new Node();
    return n;
}
```

acquire メソッドは mutex でロックを取得するまで待ち、release メソッドはロックをリリースします。

ロックによって保護されるコードの範囲が明確になるように、可能な場所に中括弧を挿入することを推奨します。

ロック用の C インターフェイスに精通している方は、なぜ mutex オブジェクト自体に単純に acquire メソッドや release メソッドがないのか疑問に思われるかもしれません。この理由は、例外が保護領域からスローされた場合、コントロールはリリースをスキップするため、C インターフェイスは例外セーフではないためです。オブジェクト指向のインターフェイスでは、保護領域が通常のコントロール・フローで終了したか例外をスローしたかに関係なく、scoped_lock オブジェクトの破棄はロックをリリースします。これは、AllocateNode の acquire メソッドおよび release メソッドを使用するバージョンでも同じです。明示的なリリースによりロックが早めにリリースされ、デストラクターはロックがリリースされたと判断して何も行いません。

インテル® スレッディング・ビルディング・ブロックのすべての mutex では同様のインターフェイスが使用されているため、習得が簡単なだけではなく、汎用的なプログラミングが可能です。例えば、すべての mutex には、入れ子の `scoped_lock` 型があります。型 *M* の mutex を指定すると、対応するロックの型は *M*`::scoped_lock` になります。

ヒント: 前の例で示したように、mutex の型には `typedef` を常に使用することを推奨します。このように、ほかのコード部分を編集することなく、後でロックの型を変更することができます。例では、`typedef` を `typedef queuing_mutex FreeListMutexType` に置換することもできます。置換後もコードは同じ動作をします。

6.1.1 Mutex の特性

mutex に精通すると、さまざまな mutex 属性を使い分けることができます。mutex 属性は、汎用性と効率性のトレードオフに関わるため、いくつか知っておくと便利です。適切な属性を選択することにより、パフォーマンスの向上に役立ちます。mutex の特性を次に説明します。

表 8: mutex の特徴と動作 の要約も参照してください。

- **スケーラブル**。一部の mutex は、*スケーラブル* と呼ばれます。mutex は実行を一度に 1 つのスレッドに制限するため、この名前は厳密には正確ではありません。*スケーラブル mutex* は、この制限*以下*のことは行いません。待機スレッドがプロセッサー・サイクルとメモリー帯域幅を過剰に消費し、実際の作業を行おうとしているスレッドの速度を減少させてしまう場合は、mutex は直列実行よりもパフォーマンスを低下させることがあります。スケーラブル mutex は、競合が少ない場合に非スケーラブルな mutex よりも低速なことがあるため、このような場合は非スケーラブルな mutex を使用する方が良いでしょう。不確かな場合は、スケーラブル mutex を使用してください。
- **フェア**。mutex は、*フェア* または *アンフェア* です。フェア mutex は、到着順にスレッドを処理し、「飢えた」スレッドを回避します。各スレッドで、それぞれ順番が回ってきます。しかし、アンフェア mutex の方が、中断のためにスリープ状態にある次のスレッドの代わりに、実行中のスレッドを先に処理するため、フェア mutex よりも高速な場合があります。
- **リエントラント**。mutex は、*リエントラント* または *非リエントラント* です。リエントラント mutex は、既にロックを保持しているスレッドが、mutex の別のロックを取得することを許可します。これは、一部の再帰アルゴリズムでは役立つものの、通常はロック実装にオーバーヘッドがかかります。
- **スリープまたはスピン**。mutex では、スレッドが*ユーザー空間でスピン* したり、待機中に*スリープ*になります。短時間の待機の場合は、スレッドをスリープ状態にするとサイクルが消費されるため、ユーザー空間でのスピンが最も速い方法です。しかし、長時間の待機の場合は、スリープはプロセッサーを必要とするほかのスレッドにプロセッサーを譲るため、スピンよりもパフォーマンスが向上します。

次は、mutex の動作の概要です。

- `spin_mutex` は、非スケーラブル、アンフェア、非リエントラントで、ユーザー空間でスピンします。一見、最悪な機能のようですが、これは*競合状態がわずかな場合*は*非常に高速*です。多くの SpinMutexes に競合状態を分散できるようなプログラムを設計できる場合は、ほかの mutex を使用してパフォーマンスを向上させることができます。mutex の競合状態が激しい場合は、いずれにしても使用しているアルゴリズムではパフォーマンスは向上しません。効率的なロックを探すよりも先に、アルゴリズムの再設計を検討してください。
- `queuing_mutex` は、スケーラブル、フェア、非リエントラントで、ユーザー空間でスピンします。これは、スケーラビリティーと公平性が重要な場合に使用します。
- `spin_rw_mutex` と `queuing_rw_mutex` は、`spin_mutex` と `queuing_mutex` に似ていますが、*リーダー*ロックをサポートします。
- `mutex` は、システムの "ネイティブな" 排他制御を行うラッパーです。Windows* システムでは、`CRITICAL_SECTION` に実装されます。Linux* システムでは、`pthread` mutex に実装されます。このラッパーを使用する利点は、例外セーフなインターフェイスを追加し、インテル® スレッディング・ビルディング・ブロックのほかの mutex と同一のインターフェイスを提供することです。これにより、のちにパフォーマンス測定を行い、必要であれば、別の mutex に簡単に切り替えることができます。

表 8: mutex の特徴と動作

Mutex	スケーラブル	フェア	リエントラント	スリープ	サイズ
`mutex`	OS 依存	OS 依存	×	○	≥ 3 ワード
`spin_mutex`	×	×	×	×	1 バイト
`queuing_mutex`	✓	✓	×	×	1 ワード
`spin_rw_mutex`	×	×	×	×	1 ワード
`queuing_rw_mutex`	✓	✓	×	×	1 ワード

6.1.2 リーダー/ライター mutex

排他制御は、少なくとも 1 つのスレッドが共有変数に *書き込み* する場合は必要ですが、複数のリーダーを保護領域に許可しても、問題はありません。mutex のリーダー /ライター型は、クラス名に _rw_ が示され、*ライターロック* と *リーダーロック* を区別することで複数のリーダーを有効にすることができます。指定された mutex には、複数のリーダーロックがある可能性があります。

scoped_lock のコンストラクター内の追加のブール・パラメーター により、リーダーロックの要求は、ライターロックの要求とは区別されます。パラメーターは、リーダーロックの要求の場合は false、ライターロックの要求の場合は true です。デフォルトは true で、省略された場合に spin_rw_mutex または queuing_rw_mutex は、_rw_ のない mutex のように動作します。次のセクションでは、リーダーロックを取得するために、パラメーターが明示的に false である例を示します。

6.1.3 アップグレード/ダウングレード

upgrade_to_writer メソッドを使用して、リーダーロックをライターロックにアップグレードすることができます。次に例を示します。

```
std::vector<string> MyVector;
typedef spin_rw_mutex MyVectorMutexType;
MyVectorMutexType MyVectorMutex;

void AddKeyIfMissing( const string& key ) {
    // Obtain a reader lock on MyVectorMutex
    MyVectorMutexType::scoped_lock lock(MyVectorMutex,/*is_writer=*/false);
    size_t n = MyVector.size();
    for( size_t i=0; i<n; ++i )
        if( MyVector[i]==key ) return;
    if( !MyVectorMutex.upgrade_to_writer() )
        // Check if key was added while lock was temporarily released
        for( int i=n; i<MyVector.size(); ++i )
           if(MyVector[i]==key ) return;
    vector.push_back(key);
}
```

ベクトルは、再度検索する必要があることに注意してください。これは、upgrade_to_writer が、アップブレードの前に一時的にロックをリリースしなければならない可能性があるためです。これを行わないと、6.1.4 の説明にあるように、デッドロックが発生することがあります。upgrade_to_writer メソッドでは、bool を返し、ロックをリリースしなくてもロックのアップグレードが成功した場合は true、一時的にロックがリリースされた場合は false です。このように、upgrade_to_writer で false が返された場合は、コードは別のライターによってキーが挿入されて

いないかをチェックする search を返します。この例では、キーが常にベクトルの最後に追加され、削除されないことが想定されています。このような想定により、ベクトル全体の検索をやり直す必要はなく、最初に検索された範囲以外の要素のみを検索することができます。重要な点は、`upgrade_to_writer` が false を返したとき、リーダーロック保持している間に確立されたすべての想定は無効になり、再確認が必要であることです。

対称的に、`downgrade_to_reader` という対応するメソッドがありますが、実際に使用する理由はあまりありません。

6.1.4 ロックの問題

ロックは、パフォーマンスの問題や精度の問題を引き起こす可能性があります。ロックを初めて使用する場合は、次の問題に注意してください。

6.1.4.1 デッドロック

デッドロックは、スレッドが複数のロックを取得しようとし、処理が必要な別のスレッドのロックをそれぞれのロックが保持しているときに発生します。より正確には、デッドロックは次のような場合に発生します。

- スレッドのサイクルがある場合。
- 各スレッドが少なくとも 1 つのロックを mutex 上で保持し、ロックを既に保持しているサイクルにある*次の*スレッドを mutex 上で待機している場合。
- どのスレッドもロックを開放しない場合。

例えば、交差点での交通渋滞を考えてみてください。それぞれの車は道路の一部を "取得" しているけれども、車が前へ進むために、別の車がいる道路の部分を "取得" する必要があります。デッドロックを回避するには、次の 2 つの一般的な方法があります。

- 同時に 2 つのロックを保持する必要がある状況を回避する。プログラムを小さなアクションに分割して、1 つのロックを保持している間にそれぞれが処理を終了できるようにします。
- 常に同じ順番でロックを取得する。例えば、"外部コンテナー" mutex と "内部コンテナー" mutex があり、それぞれでロックが 1 つ必要な場合は、常に外側が先にロックを取得するようにします。また、別の例をあげると、ロックに名前があり、"アルファベット順にロックを取得" する場合です。または、ロックに名前がない場合は、mutex の数値アドレス順でロックを取得します。

- 後のセクションで説明されているように、ロックの代わりにアトミック操作を使用してください。

6.1.4.2 コンボイ

もう 1 つのロックの問題は *コンボイ* です。コンボイは、ロックを保持しているスレッドがオペレーティング・システムにより中断されたときに発生します。その他すべてのスレッドは、中断されたスレッドが再開し、ロックをリリースするまで待機しなければなりません。フェア mutex では、待機スレッドが中断された場合にその後続のすべてのスレッドが待機しなければならないため、状況を悪化させてしまいます。

コンボイを最小限に抑えるためには、ロックはできるだけ短時間でリリースするようにします。また、ロックを取得する前にできるだけ事前に処理しておきます。

コンボイを回避するため、できる限りロックの代わりにアトミック操作を使用してください。

7 アトミック操作

アトミック操作を使用して、排他制御の問題を回避することができます。スレッドがアトミック操作を行うと、その他のスレッドではその操作が瞬時に発生したかのように認識されます。アトミック操作の長所は、ロックに比べて比較的処理が速く、デッドロックやコンボイなどを回避できる点にあります。短所は、限られた操作しか行うことができず、より複雑な操作を効率良く総合的に扱うには不十分なことです。しかし、排他制御の代わりにアトミック操作を使用できる機会を逃す手はありません。atomic<*T*> クラスは、C++ スタイルでアトミック操作を実装します。

アトミック操作の典型的な使用方法は、スレッドセーフな参照カウントに使用することです。x が int 型の参照カウントで、プログラムでは参照カウントが 0 になったときに何らかのアクションが必要だとします。シングルスレッド・コードでは、x に普通の int を使用し、`--x; if(x==0) action()` を書き込みます。しかし、この方法は、次の表で示されるように、2 つのスレッドが自身の操作をインターリーブするため、マルチスレッド・コードでは失敗する可能性があります。t_a と t_b はマシンレジスターで、時間は下方向に経過します。

表 9: 機械語命令のインターリーブ

スレッド A	スレッド B
t_a = x	
	t_b = x
x = t_a - 1	
	x = t_b - 1
if(x==0)	
	if(x==0)

コードでは x が 2 回デクリメントされることになっていますが、値は、元の値と比べて 1 つしか減っていません。また、*x* のテストがデクリメントとは別で行なわれるためにほかの問題も生じます。*x* が 2 から開始して、どちらかのスレッドが if 条件を評価する前に、両方のスレッドで *x* 回デクリメントする場合、*両方*のスレッドが action() を呼び出してしまいます。この問題を解決するには、1 つのスレッドのみがデクリメントを行うようにし、*さらに*"if" によってチェックされる値がデクリメントの結果の値であることを保証しなければなりません。これは、mutex の導入により行うことができますが、*x* を atomic<int> として宣言し、"`if(--x==0) action()`" にすることによって、より素早く簡単に行うことができます。atomic<int>::operator-- メソッドはアトミックに動作します。ほかのスレッドは介入できません。

atomic<*T*> は、整数型またはポインター型の *T* 型でアトミック操作をサポートします。5 つの基本的な操作がサポートされており、構文の便宜上、オーバーロードされた演算子の形式のインターフェイスが追加されています。例えば、atomic<*T*> の ++、--、-=、+= 演算子は、基本操作 *fetch-and-add* の形式です。次の表は、atomic<*T*> 型の変数 *x* の 5 つの基本操作です。

表 10: atomic<T> 型の変数 x の基本操作

= *x*	*x* の値を読み取ります。
x =	*x* の値を書き込み、値を返します。
x.fetch_and_store(y)	*y*=*x* を行い、*x* の古い値を返します。
x.fetch_and_add(*y*)	*x*+=*y* を行い、*x* の古い値を返します。
x.compare_and_swap(*y*,*z*)	*x* が *z* に等しい場合は、*x*=*y* を行います。いずれの場合でも、*x* の古い値を返します。

これらの操作はアトミックに行われるため、排他制御がなくても安全に使用することができます。次の例について考えてみます。

```
atomic<unsigned> counter;

unsigned GetUniqueInteger() {
    return counter.fetch_and_add(1);
}
```

GetUniqueInteger ルーチンは、カウンターが巡回するまで呼び出されるごとに異なる整数を返します。どれだけのスレッドが同時に GetUniqueInteger ルーチンを呼び出してもこれは変わりません。

compare_and_swap 操作は、多くの非ブロック・アルゴリズムにおいて基本操作です。排他制御での問題は、ロックを保持しているスレッドが停止した場合に、ほかのすべてのスレッドは、保持スレッドが再開されるまでブロックされる点にあります。非ブロック・アルゴリズムでは、ロックではなく、アトミック操作を使用することによってこの問題を回避します。アトミック操作は、一般に複雑で、確認には洗練された解析が必要ですが、次の慣用句は、わかりやすく、また知っておくと良いでしょう。この慣用句は共有変数 globalx を古い値を基にして更新します。

```
atomic<int> globalx;

int UpdateX() {     // Update x and return old value of x.
    do {
        // Read globalX
```

```
        oldx = globalx;
        // Compute new value
        newx = ...expression involving oldx....
        // Store new value if another thread has not changed globalX.
    } while( globalx.compare_and_swap(newx,oldx)!=oldx );
    return oldx;
}
```

さらに、いくつかのスレッドは、ほかのスレッドが干渉しなくなるまでループを繰り返します。通常、更新に、2、3 の命令しか必要がない場合は、この慣用句に対応する排他制御よりも処理が迅速です。

注意: 次のシーケンスが意図に反する場合は、この慣用句は適切ではありません。

スレッドが `globalx` の *A* の値を読み取る

ほかのスレッドが `globalx` を *A* から *B* へ、そして再び、*A* へに変更する

ステップ 1 のスレッドは、自身の `compare_and_swap` を行い、*A* を読み取るため、*B* になされた変更を認識しません。

この問題は、*ABA 問題* と呼ばれます。リンクされたデータ構造に対する非ブロック・アルゴリズムの設計に問題があるケースが大半を占めます。詳細は、インターネットで検索してください。

7.1.1 atomic<T> にコンストラクターがない理由

`atomic<T>` テンプレート・クラスには、意図的にコンストラクターが用意されていません。第 7 章の `GetUniqueInteger` の例のように、すべてのファイルスコープ・コンストラクターが呼び出される前であっても、正常に動作する必要があるためです。`atomic<T>` にコンストラクターがあると、ファイルスコープ・インスタンスが参照された後で初期化される可能性があります。`atomic<T>` を 0 に初期化する場合はゼロ初期化を使用できます。

7.1.2 メモリー一貫性

Itanium® アーキテクチャーなど、"メモリー一貫性の弱い" 一部のアーキテクチャーでは、効率性のためにハードウェアによって異なるアドレスのメモリー操作の順番が変更される場合があります。この複雑な動作については、資料 (Intel 2002, Robison 2003) をご覧ください。IA-32 アーキテクチャー・プラットフォームおよび インテル® 64 アーキテクチャー・プラットフォームのみでプログラミングを行う場合は、このセクションをスキップしても問題ありません。

順序の変更を制限するするために、プロセッサーは、*フェンス* を起動することがあります。フェンスの種類については、表 11: フェンスの種類 の説明を参照してください。

表 11: フェンスの種類

種類	説明	デフォルト
`acquire`	**フェンスの後の操作はフェンスを決して越えません。**	**read**
`release`	**フェンスの前の操作はフェンスを決して越えません。**	**write**

full	**フェンスの両側の操作はフェンスを決して越えません。**	`fetch_and_store`
		`fetch_and_add`
		`compare_and_swap`

「デフォルト」の列は、デフォルト操作を示します。これらのデフォルトを使用して、予期しない問題を回避します。読み取り/書き込みの場合、デフォルトは利用可能なフェンスの種類のみです。ただし、弱いメモリー一貫性に精通している場合は、フルフェンスをより小さなフェンスに変更することもできます。これを行うには、テンプレート引数をとる可変要素を使用します。引数は `acquire` または `release` で、enum 型のメモリー・セマンティクスの値です。

例えば、さまざまなスレッドがデータ構造体の要素を生成し、データ構造が準備できたときに消費スレッドにシグナルを発信させるとします。これを行う 1 つの方法としては、ビジーな生成スレッドの数でアトミックカウンターを初期化し、各生産スレッドが終了したら、次の操作を行います。

```
refcount.fetch_and_add<release>(-1);
```

`release` 引数は、refcount がデクリメントされる前に生産スレッドの共有メモリーへの書き込みが行われることを保証します。同様に、消費スレッドが `refcount` をチェックするときは、消費スレッドのデータ構造の読み取りが、`refcount` が 0 になるのを見届けた後で行えるように、`acquire` フェンス (読み取りのデフォルト) を使用しなければなりません。

8 時間計測

並列プログラムのパフォーマンスは、通常、CPU 時間ではなく、*ウォールクロック* 時間で測定します。これは、並列化が進むにつれ、より多くの CPU を使用し、CPU の総時間が増加するためです。プログラムの並列化の最終目的は、リアルタイムでプログラムを*より速く*実行することです。

インテル® スレッディング・ビルディング・ブロックの tick_count クラスは、ウォールクロック時間を測定するための単純なインターフェイスを提供します。tick_count の値は、現在の絶対時間を表すスタティック・メソッド tick_count::now() から取得されます。2 つの tick_count 値を引くと、tick_count::interval_t で相対時間値が求められます。この値は次の例のように、秒に変換することができます。

```
tick_count t0 = tick_count::now();
... do some work ...
tick_count t1 = tick_count::now();
printf( “work took %g seconds\n” ,(t1-t0).seconds());
```

一部の時間計測インターフェイスとは異なり、tick_count は複数のスレッドにわたって使用しても安全であることが保証されています。異なるスレッドで作成された tick_count 値の差分を求められます。tick_count の差は秒に変換することができます。

tick_count の分解能は、同一プロセス内の複数のスレッド間で有効な、プラットフォームで最高の分解能タイミング サービスに対応します。CPU タイマーレジスターは、プラットフォームによっては、複数のスレッド間で有効 *ではない* ため、tick_count の分解能が別のプラットフォームと一致している保証はありません。

9 メモリー割り当て

インテル® スレッディング・ビルディング・ブロックには、STL テンプレート・クラス std::allocator に似た 2 つのメモリー・アロケーター・テンプレートが用意されています。scalable_allocator<T> と cache_aligned_allocator<T> の 2 つのテンプレートは、次のように、並列プログラムのクリティカルな問題に対処します。

- **スケーラビリティー:** スケーラビリティーの問題は、元々シリアルプログラム向けに設計されたメモリー・アロケーターを、スレッドで使用する際に発生します。スレッドは、1 回に 1 つのスレッドしか割り当てができない方法で、シングル共有プールで競合します。scalable_allocator<T> メモリー・アロケーター・テンプレートを使用して、そのようなスケーラビリティーのボトルネックを回避します。このテンプレートは、メモリーを迅速に割り当ててリリースし、プログラムのパフォーマンスを向上させます。
- **フォールス・シェアリング**: シェアリングの問題は、2 つのスレッドが、同一キャッシュラインを共有する別々のワードにアクセスする際に生じます。問題は、キャッシュラインがプロセッサー・キャッシュ間の情報交換単位であるということです。1 つのプロセッサーがキャッシュラインを変更し、別のプロセッサーが同じキャッシュラインの読み取り (または書き込み) を行う場合、たとえ、2 つのプロセッサーがライン中の異なるワードを処理していたとしても、キャッシュラインはプロセッサーから別のプロセッサーへ移動しなければなりません。フォールス・シェアリングは、キャッシュラインの移動に何百ものクロックが必要なため、パフォーマンスに影響します。

cache_aligned_allocator<T> クラスを使用して、常にキャッシュライン上で割り当てます。cache_aligned_allocator によって割り当てられる 2 つのオブジェクトでは、フォールス・シェアリングが発生しないことが保証されます。オブジェクトが cache_aligned_allocator に割り当てられ、もう 1 つのオブジェクトが別の方法で割り当てられた場合は、保証されません。cache_aligned_allocator へのインターフェイスは、std::allocator と同一なため、これを STL テンプレート・クラスへの *allocator* 引数として使用することができます。

次のコードは、cache_aligned_allocator を使用する STL ベクトルを宣言して割り当てる方法を示しています。

```
std::vector<int,cache_aligned_allocator<int> >;
```

ヒント: cache_aligned_allocator<T> 機能は、たとえ小さなオブジェクトであっても、メモリーに対応するキャッシュラインを少なくとも 1 つ割り当てる必要があるため、空間を多少犠牲にします。そのため、フォールス・シェアリングが重要な問題になりそうな場合のみ、cache_aligned_allocator<T> を使用するようにしてください。

スケーラブル・メモリー・アロケーターは、インテルの PSL CTG チームによって開発された McRT テクノロジーを採用しています。

9.1 使用するダイナミック・ライブラリー

2.2 で説明されているように、scalable_allocator<T> テンプレートでは、インテル® スレッディング・ビルディング・ブロックのスケラーブル・メモリー・アロケーター・ライブラリーが必要です。これは、インテル® スレッディング・ビルディング・ブロックの汎用ライブラリーは必要なく、また残りのインテル® スレッディング・ビルディング・ブロックとは独立して使用することが可能です。

cache_aligned_allocator<T> テンプレートは、スケーラブル・アロケーター・ライブラリーがある場合はこれを使用し、ない場合は malloc と free を使用します。このように、cache_aligned_allocator<T> は、スケラーブル・メモリー・アロケーター・ライブラリーを省略するアプリケーションでも使用できます。

残りのインテル® スレッディング・ビルディング・ブロックは、インテル® スレッディング・ビルディング・ブロックのスケラーブル・メモリー・アロケーター・ライブラリーの有無に関わらず使用できます。

テンプレート	要件	メモ
scalable_allocator<T>	インテル® スレッディング・ビルディング・ブロックのスケラーブル・メモリー・アロケーター・ライブラリー。2.2 を参照してください。	
cache_aligned_allocator<T>		スケーラブル・アロケーター・ライブラリーがある場合はそれを使用し、ない場合は malloc と free を使用します。

10 タスク・スケジューラー

このセクションでは、インテル® スレッディング・ビルディング・ブロックの*タスク・スケジューラー*について説明します。タスク・スケジューラー は、ループ・テンプレートを強化したエンジンです。通常は、タスク・スケジューラーの代わりにループ・テンプレートを使用してください。テンプレートによりスケジューラーの複雑性が隠避されます。ただし、このハイレベルなテンプレートで自然なマップができないアルゴリズムの場合は、タスク・スケジューラーを使用してください。すべてのスケジューラーの機能がこのハイレベルなテンプレートを通して直接使用できるため、既存のテンプレートと同程度の強力な新しいハイレベルなテンプレートをビルドすることができます。

10.1 タスクベースのプログラミング

パフォーマンスが重要な場合、マルチスレッド・プログラミングを行う方法として、スレッドのプログラミングは最適ではないことがあります。スレッドではなく、*論理タスク* の観点からプログラムを構築したほうが非常に有効です。これには、次のようのないくつかの理由があります。

1. 利用可能なリソースへの並列化のマッチング
2. タスクの迅速なスタートアップとシャットダウン
3. より効率的な評価順
4. ロード・バランシングの向上
5. よりハイレベルでの思考

次に、上記のポイントを詳細に説明します。

スレッドパッケージを使用して作成するスレッドは、*論理* スレッドで、ハードウェアの*物理スレッド*にマップします。外部デバイスで待機しない計算の場合、最も効率が良いのは、1 つの物理スレッドで 1 つの論理スレッドが実行されているときです。それ以外は、ミスマッチから効率性が低下します。*アンダーサブスクリプション* は、物理スレッドが動作し続けるのに十分な論理スレッドが実行されていない場合に発生します。*オーバーサブスクリプション* は、物理スレッドよりも多い論理スレッドが実行されているときに発生します。通常、オーバーサブスクリプションは、論理スレッドの*タイムスライス* 実行を引き起こし、付録 A の タイムスライスのコスト で説明されているような、オーバーヘッドを発生させます。スケジューラーは、物理スレッドごとに 1 つの論理スレッドを割り当て、同じプ

ロセスまたは別のプロセスのスレッドからの干渉を許可し、タスクを論理スレッドにマッピングすることにより、オーバーサブスクリプションを回避しようとします。

論理スレッドに比べてタスクの方が良い点は、タスクは*非常に軽い*ことです。Linux システムでは、タスクの開始と終了は、スレッドの開始と終了よりも 18 倍も速く処理されます。Windows システムでは、その比率が 100 以上になります。これは、スレッドにはレジスターやスタックなどの多くのリソース自身のコピーがあるためです。Linux では、スレッドにはプロセス id もあります。一方、インテル® スレッディング・ビルディング・ブロックのタスクは、比較的小さなルーチンで、また タスクレベルではプリエンプトできません (論理スレッドのプリエンプトは可能)。

インテル® スレッディング・ビルディング・ブロックのタスクも、*スケジューラー がアンフェア*であるため、効率性が高くなります。スレッド・スケジューラーは、通常、ラウンドロビン方式でタイムスライスを分配します。この分配は、各論理スレッドに公平に時間が配分されるため "フェア" と呼ばれます。プログラムのハイレベルな編成がわからなくても使用できる最も安全な方法であるため、スレッド・スケジューラーは、通常フェア (公平) です。タスクベースのプログラミングでは、タスク・スケジューラーはよりハイレベルな情報を持つため、効率性の観点から公平性を犠牲にできます。実際、タスク・スケジューラーは、効果が得られるように、タスクの開始を遅らせることが多くあります。10.3 では、どのような効果があるか、また、どのように時間と空間が節約されるかを説明します。

スケジューラーは、*ロード・バランシング*を行います。正しいスレッドの数を使用することに加え、それらのスレッドに均等に作業を分配することが重要です。プログラムが十分に小さなタスクに分割されている限り、スケジューラーは負荷のバランスをとりながらスレッドにタスクを割り当てることができます。スレッドベースのプログラミングでは、ロード・バランシングをプログラマー自身で処理しなければならないため、正しく行うことが困難になります。

最後に、スレッドの代わりにタスクを使用する主な利点は、タスクでは、よりハイレベルな、タスクベースのレベルで考えられることです。スレッドベースのプログラミングでは、効率性の観点から低レベルの物理スレッドで考えなければなりません。これは、アンダーサブスクリプションやオーバーサブスクリプションを防ぐためには、物理スレッドごとに 1 つの論理スレッドしか割り当てられないためです。さらに、比較的粗粒度のスレッドを処理する必要もあります。タスクを使用すると、効率の良いスケジューリングはスケジューラーに任せて、プログラマーはタスク間の論理的な依存性に集中することができます。

10.1.1 タスクベースのプログラミングが適切ではない場合

タスク・スケジューラーの使用は、パフォーマンスの向上を目的としたスレッド化アプローチとしては最も良い方法ですが、タスク・スケジューラーの使用が適切ではない場合もあります。タスク・スケジューラーは、非ブロッキング・タスクから構成されるハイパフォーマンスなアルゴリズムを意図していますが、タスクがまれにブロックする場合でも動作します。ただし、スレッドが頻繁にブロックする場合は、スレッドがブロックされている間は何の作業も行えないため、タスク・スケジューラーを使用する際にパフォーマンス・ロスが発生します。ブロッキングは、I/O や mutex のために長時間待機する間に発生します。スレッドが、長い間 mutex を保持する場合は、どれだけ多くのスレッドがあったとしてもコードのパフォーマンスは良くありません。ブロッキング・タスクがある場合は、完璧なスレッドを使用すると良いでしょう。タスク・スケジューラーは、作成したスレッドとインテル® スレッディング・ビルディング・ブロックのタスクを安全に混在できるように設計されています。

10.2 簡単な例: フィボナッチ数

このセクションでは、例として *n* 番目のフィボナッチ数を計算します。この例では、非効率的なメソッド [1] を使用してフィボナッチ数を計算しますが、単純な再帰パターンを使用するタスク・ライブラリーの基本も示しています。タスクベースのプログラミングからスケーラブルな速度の向上を得るには、多くのタスクを指定する必要があります。これは、通常、再帰的なタスクパターンを使用して、インテル® スレッディング・ビルディング・ブロックで行えます。

シリアルコード:

```
long SerialFib( long n ) {
    if( n<2 )
        return n;
    else
        return SerialFib(n-1)+SerialFib(n-2);
}
```

並列タスクベースのバージョンのトップレベル・コード:

```
long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(task::allocate_root()) FibTask(n,&sum);
```

---

[1] 効率的なメソッドは、$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}$ を計算して、左上の要素をとる方法です。繰り返し 2 乗法を使用すると累乗を素早く行うことができます。

```
    task::spawn_root_and_wait(a);
    return sum;
}
```

このコードは、実際の作業を行うのに FibTask 型のタスクを使用します。次のようなステップで行います。

1. タスクに空間を割り当てます。これは、特別な”オーバーロードされた new” と task::allocate_root メソッドで行われます。名前の _root サフィックスは、作成されたタスクには親がいないことを示します。このタスクは、タスクツリーのルートです。タスクの終了時に空間を効率良く再利用できるように特別なメソッドによってタスクを割り当てる必要があります。
2. new により起動される FibTask(n,&sum) コンストラクターでタスクを構築します。ステップ 3 のタスクの実行時に、n 番目のフィボナッチ数を計算して、*sum に格納します。
3. task::spawn_root_and_wait でタスクの完了を実行します。

実際の作業は、FibTask 構造の中にあります。次にその定義を示します。

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    task* execute() {             // Overrides virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
            // Set ref_count to "two children plus one for the wait".
            set_ref_count(3);
            // Start b running.
            spawn( b );
            // Start a running and wait for all children (a and b).
            spawn_and_wait_for_all( a );
            // Do the sum
            *sum = x+y;
        }
        return NULL;
    }
};
```

標準 C++ の拡張機能を使用せずに、並列化を表現するため、SerialFib よりも比較的大きなコードです。

インテル® スレッディング・ビルディング・ブロックによってスケジューリングされるすべてのタスクと同様に、FibTask は task クラスから派生しています。n フィールドと sum フィールドは、それぞれ入力値と出力へのポインターを保持します。これらは、FibTask のコンストラクターに渡される引数のコピーです。execute メソッドは、実際の計算を行います。すべてのタスクでは、純粋な仮想メソッド task::execute を変更する execute の定義を提供しなければなりません。定義は、タスクの作業を行い、NULL または次に実行するタスクへのポインターを返します。この例では、NULL を返します。NULL ではない場合については、セクション 10.4.3 で説明します。

FibTask::execute() メソッドは次の処理を行います。

- n が小さく、直列実行の方が高速かどうかをチェックします。CutOff の適切な値を見つけるには、試行が必要です。少なくとも 16 が、この例で最も速度向上を見込める値です。問題サイズが小さくなったときに、シーケンシャルなアルゴリズムに並べ替えることは、並列化の分割統治法の特性です。切り替えのポイントを見つけるには試行が必要なため、実験できるようなコードを記述するようにしてください。
- else がとられる場合、コードは (n-1) 番目と (n-2) 番目のフィボナッチ数の計算を行う 2 つの子タスクを実行します。ここで、継承メソッドの allocate_child() がタスクの空間割り当てに使用されます。トップレベルのルーチン ParallelFib が allocate_root() を使用してタスクに空間を割り当てたことを思い出してください。違いは、ここではタスクが *子* タスクを作成しているということです。この関係は、割り当てメソッドの選択によって示されます。
- set_ref_count(3) を呼び出します。数字の 3 は、2 つの子と spawn_and_wait_for_all メソッドで必要な追加の暗黙的参照を表現しています。子を生成する前に、set_reference_count(3) を必ず呼び出してください。呼び出しが行われない場合は、未定義の動作が発生します。ライブラリーのデバッグバージョンは、通常この種のエラーを検出し、レポートします。
- 2 つの子タスクを生成します。タスクの生成は、選択したときにいつでも、あるいは別のタスクを実行しながらでも、並行にそのタスクを実行できることをスケジューラーに示唆します。実行ポリシーについては、セクション 10.3 で説明します。spawn メソッドによる 1 番目の子の生成では、子タスクが実行を開始するのを待つことなくすぐにリターンします。spawn_and_wait_for_all メソッドによる 2 番目の子の生成では、親は現在割り当てられている子のタスクが終了するまで待機します。
- 2 つの子が終了した後で、親は x+y を計算して、*sum に結果を格納します。

タスクでは 2 つの子タスクしか生成されないため、一見、並列化に制約があるように見えます。ここでは、*再帰的並列化* がポイントです。2 つの子タスクは、それぞれ 2 つの子タスクを生成し、その後も `n<Cutoff` に到達するまでこの作業を繰り返します。この連鎖反応は、多くの潜在的な並列化の機会を作り出します。タスク・スケジューラーの利点は、この潜在的な並列化の機会を、非常に効率的な方法で実際の並列化に実現させられることです。これは、比較的わずかなコンテキストの切り替えで物理スレッドがビジー状態になるように、実行するタスクが選択されるためです。

10.3 タスク・スケジューリングの動作

スケジューラーは、*タスクグラフ* を評価します。グラフは、ノードがタスクを表す有向グラフで、それぞれ、*親* を指しています。親は自身の終了を待機している別のタスクか、または NULL です。`task::parent()` メソッドは、親ポインターへの読み取り専用アクセスをユーザーに与えます。各タスクには、そのタスクを親として持つタスク数をカウントする *refcount* があります。また、各タスクにはその親の深さよりも一段深い、*深さ* があります。次の図は、フィボナッチ例のタスクグラフを示しています。

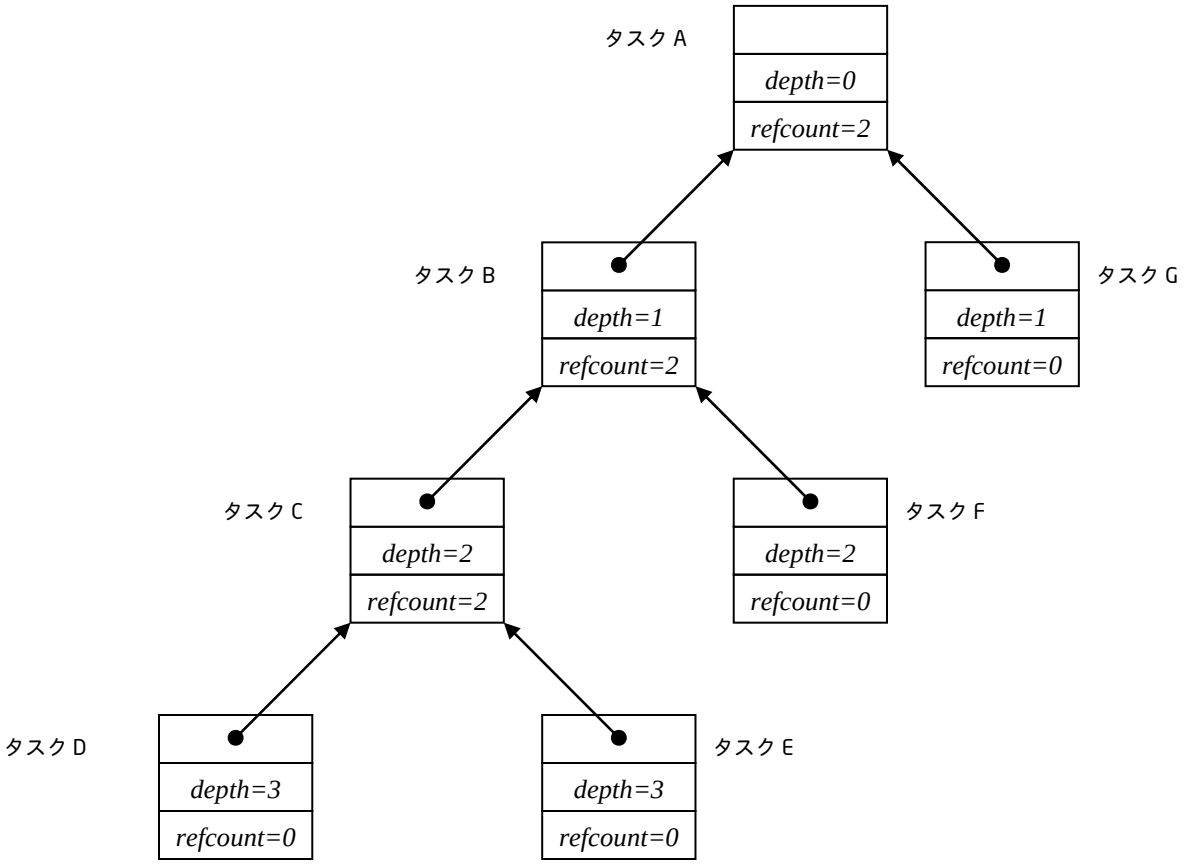


図 4: フィボナッチ例のタスクグラフ

図では、非ゼロ参照カウント (A、B、C) がその子タスクの終了を待機しています。リーフタスクは実行中か、あるいは実行準備ができている状態です。

スケジューラーは、メモリー需要とクロススレッド通信の両方を最小限に抑えるような方法でタスクを実行します。これには、深さ優先の実行と幅優先の実行のバランスをとる必要があります。ツリーが有限であると仮定すると、シーケンシャル実行には深さ優先が最も良い方法です。これには、次のような理由があります。

- **キャッシュは熱い (ホット) うちに打つ**。最も深いタスクは、最も新しく生成されたタスクで、キャッシュの中で一番ホットな状態です。また、それらのタスクが終了できる場合、タスク C は実行を続けることができ、キャッシュで最もホットではないにしても、その上にある古いタスクよりはホットな (新しい) 状態です。
- **空間を最小限に抑える**。最も浅いタスクを実行すると、幅優先のツリー展開になります。これにより、同時に共存できる、指数関数的な数のノードが作成されます。一方、深さ優先の実行では

同じ数のノードが作成されますが、ほかの準備ができているタスク (図では、E、F、G) をスタックしてしまうために、直線の数のみが同時に共存できます。

幅優先の実行にはメモリー消費の深刻な問題がありますが、物理スレッドの数が無限の場合は、並列化を最大限に引き出す実行方法です。物理スレッドは限られているため、利用可能なプロセッサーを常にビジーな状態にしておける分だけの幅優先の実行を使用すると良いでしょう。スケジューラーは、次のような幅優先の実行を実装します。

- 各スレッドには、自身の*レディープール* (タスクの配列リスト) があります。
- 実行準備ができたとみなされると、タスクはプールに移動します。
- 各スレッドは、必要なときにほかのプールからタスクをスチールします。[2]

図 5 は、図 4 のタスクグラフに対応するプールの図を示しています。

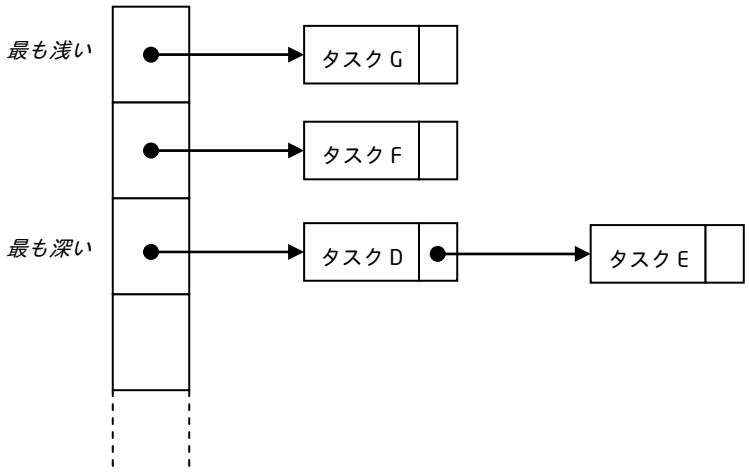


図 5: スレッドのレディープール

このプールは、リストの配列で構成されています。配列は、タスクの深さによって添字されます。リストは、スタックとして扱われます。タスクは、リストの左側に押し出され、同様に左側からポップされます。それぞれのレディープールでは、タスクをプールに入れるアクションと、タスクを実行するためにプールからタスクを取り出すアクション の 2 つのアクションが起こります。

---

[2] タスク・スケジューラーは、初期の Cilk スケジューラーの影響を受けています。Cilk スケジューラーについては、『Cilk: An Efficient Multithreaded Runtime System", PPoPP 95』 (英語) を参照してください。

スレッドがタスクグラフ評価に入り、新しいタスクを実行する必要があるときに、次の規則に一番最初に当てはまるスレッドがタスクを取得します。

1. 前のタスクの `execute` メソッドによって返されたタスクを使用する。この規則は、`execute` が `NULL` を返した場合には適用されません。
2. *自身の* プールの *最も深い* リストの先頭にあるタスクをとる。この規則は、プール内にあるすべてのリストが空の場合には適用されません。
3. 無作為に選択された *別の* プールの *最も浅い* リストの先頭からスチールする。選択されたプールが空の場合は、スレッドはこの規則が当てはまるまで実行します。

取得は常に自動で、タスクグラフ評価の一部として行われます。配置は明示的または自動です。

ここでは、タスクをレディープールに配置する方法を紹介します。タスクは、常に配置スレッドのレディープールに移動します。別のプールからのスチールは許可されますが、別のプールへの寄付はできません。

タスクをレディープールに配置するには 3 つの方法があります。

- タスクが `spawn` メソッドなどによって明示的に作成される。
- `task::recycle_to_reexecute` メソッドでタスクに再実行のマークが付けられている。
- 子タスクの終了時に暗黙的にデクリメントされた後でタスクの参照カウントがゼロになる。これは、最後の子タスクが完了したときに常に起こるとは限りません。これは、たまにタスクの自動作成が望ましくない場合に、仮想的な "ガード参照" が追加されるためです。

要約すると、タスク・スケジューラーの基本的な手法は、"幅優先のスチールと深さ優先の作業" です。幅優先のスチール規則は、スレッドが十分にビジーであるような並列化をもたらします。幅優先の作業規則では、いったん十分な作業を取得してしまえば、スレッドがそれを効率良く操作できるようにします。

10.4 役立つタスク手法

このセクションでは、スケジューラーを最大限に活用するためのプログラミング手法を説明します。

10.4.1 再帰連鎖反応

スケジューラーは、ツリー構造のタスクグラフで最も良く動作します。これは、"幅優先のスチールと深さ優先の作業" 手法が最も良く適用できるためです。また、ツリー構造のタスクグラフでは、多くのタスクを迅速に生成することができます。例えば、マスタータスクが *N* 個の子を直接生成しようとすると、O(*N*) ステップが必要です。しかし、ツリー構造のフォークでは、O(lg(*N*)) ステップしか必要ありません。

ドメインが明らかにツリー構造ではないことがよくありますが、簡単にツリーにマップすることができます。例えば、`parallel_for` (`tbb/parallel_for` 内) は、整数シーケンスなど、反復空間で動作します。セクション 3.4 では、2 つの反復空間に分割する方法で反復空間がどのように定義されているかを示します。`parallel_for` テンプレート関数は、反復空間をバイナリーツリーに再帰的にマップするためにその定義を使用します。

10.4.2 継続渡し

`spawn_and_wait_for_all` メソッドは、子タスク上で待機する便利な方法ですが、スレッドがアイドル状態になると少々非効率的です。アイドルスレッドは、別のスレッドからタスクをスチールすることによってビジー状態を保とうとします。スケジューラーは、可能性のある犠牲タスクを、待機タスクよりも深いところにあるタスクに制限します。この制限によって、最も浅いタスクが選ばれるというポリシーは変更されます。この制限は、最悪の場合にメモリー需要を抑えます。この制限を回避する方法は、親を待機させずに、単純に子と return の両方を生成するようにすることです。子は、その親の子としてではなく、親の *継続タスク* (両方の子が完了した時点で実行されるタスク) として割り当てられます。`FibTask` の "継続渡し" 可変要素を次に示します。変更された部分は、青色で示されています。

```
struct FibContinuation: public task {
    long* const sum;
    long x, y;
    FibContinuation( long* sum_ ) : sum(sum_) {}
    task* execute() {
        *sum = x+y;
        return NULL;
```

```
    }
};

struct FibTask: public task {
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            // Set ref_count to "two children plus one for the wait".
            c.set_ref_count(2~~3~~);
            c.spawn( b );
            c.spawn( a );
            return NULL;
        }
    }
};
```

次のオリジナル・バージョンと継続渡しバージョンの違いを理解しておく必要があります。

大きな違いは、オリジナル・バージョンでは x と y が execute メソッドでローカル変数だということです。継続渡しバージョンでは、子が終了する前に親がリターンするため、これらはローカル変数ではありえません。その代わり、継続タスク FibContinuation のフィールドになります。

割り当てロジックは変更されます。継続バージョンは allocate_continuation で割り当てられます。これは、継続の深さが、子が親よりも一段深い場所ではなく、親と同じであることを除き allocate_child と似ています。また、これは this の*親*を c に進め、this の*親*を NULL に設定します。次の図は、この変換をまとめています。

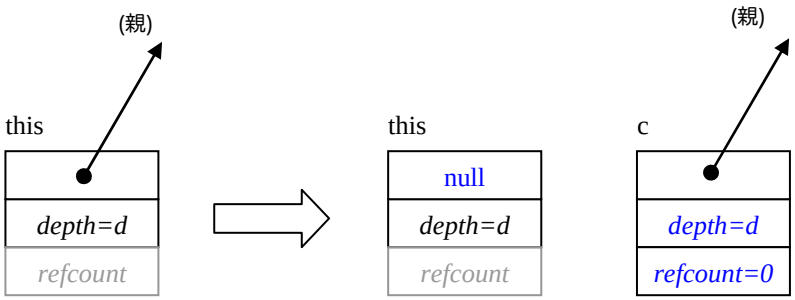


図 6: `allocate_child` のアクション

変換のプロパティーは、親の参照カウントを変更しないため、参照カウントロジックによる干渉を回避できます。

参照カウントは、子の数の 2 に設定されます。オリジナル・バージョンでは、`spawn_and_wait_for_all` では、増分カウントが必要なため、3 に設定されています。さらに、コードは、子で待機するのは継続の実行のため、親ではなく、継続の参照カウントを設定します。

`sum` ポインターは、コンストラクターによって継続に渡されます。これは、`*sum` に格納するのは、現在は `FibContinuation` であるためです。子は、まだ `allocate_child` によって割り当てられますが、親ではなく、継続 `c` の子として割り当てられること注意してください。これは、`this` ではなく、`c` が、両方の子が完了したときに自動的に生成された子の "従属 (dependent)" になるためです。誤って `this.allocate_child()` を使用した場合は、両方の子が完了した後で、親タスクが再び実行されます。

オリジナルのトップレベル・コード、`ParallelFib` がどのように記述されていたかを考えてみると、ルートの `FibTask` が子の完了前に完了し、そしてトップレベル・コードは ルートの `FibTask` で待機するのに `spawn_root_and_wait` を使用しているため、継続渡しスタイルがコードを分割するのではないかと不安になるかもしれません。しかし、`spawn_root_and_wait` は継続渡しスタイルで正常に動作するように設計されているため、問題ではありません。`spawn_root_and_wait(`*x*`)` の起動は、実際には *x* の終了を待ちません。その代わり、*x* の仮の従属を作成して、その従属の参照カウントがデクリメントするのを待ちます。`allocate_continuation` は、この仮の従属を継続に転送するため、継続が終了するまで仮の従属の参照カウントはデクリメントされません。

10.4.3 スケジューラーのバイパス

スケジューラーのバイパスは、次に実行するタスクを直接指定することにより向上を図ることができます。継続渡しスタイルは、スケジューラー・バイパスの機会を広げることがあります。例えば、継続渡しスタイルの例で、10.3 の"取得" 規則のように、いったん、FibTask::execute() が返ってしまえば、タスク "a" がレディープールから取得される次のタスクになります。レディープールにタスクを配置し、その後そのタスクを取得し戻すと、オーバーヘッドが生じます。これは回避することが可能です。回避するには、execute() メソッドでタスクを生成せずに、結果としてタスクへのポインターを返します。次の例は、必要な変更を示しています。

```
struct FibTask: public task {
    ...
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            // Set ref_count to "two children".
            set_ref_count(2);
            c.spawn( b );
            c.spawn( a );
            return NULL;
            return &a;
        }
    }
};
```

10.4.4 再利用

スケジューラーをバイパスできるだけではなく、タスク割り当てと割り当て解除もバイパスすることができます。このような機会は、スケジューラー・バイパスを行う再帰タスクにあります。これは、親が完了した時点でその子が、return 文によって処理されるためです。次のコード例は、スケジューラー・バイパスの例で、再利用を実装するために必要な変更を示しています。

```
struct FibTask: public task {
    const long n;
    long* const sum;
    ...
    task* execute() {
        if( n<CutOff ) {
            *sum = SerialFib(n);
            return NULL;
```

```
        } else {
            FibContinuation& c =
                *new( allocate_continuation() ) FibContinuation(sum);
            FibTask& a = *new( c.allocate_child() ) FibTask(n-2,&c.x);
            FibTask& b = *new( c.allocate_child() ) FibTask(n-1,&c.y);
            recycle_as_child_of(c);
            n -= 2;
            sum = &c.x;
            // Set ref_count to "two children".
            set_ref_count(2);
            c.spawn( b );
            return &a;
            return this;
        }
    }
};
```

a と呼ばれていた子は、現在は、再利用された this です。recycle_as_child_of(c) 呼び出しには、いくつかの効果があります。

- execute() が返ったときに自動で破棄されないように this をマークします。
- this の深さが c よりも一段深くなるように設定します。
- this の従属を c に設定します。参照カウントにおける問題を防ぐため、this には NULL 従属がなければならないという前提条件が recycle_as_child_of にあります。これは、allocate_continuation が発生した後の場合です。

再利用時には、オリジナルのタスクフィールドが、タスクが実行開始した後で使用されないようにする必要があります。この例では、スケジューラー・バイパスを使用してこれを行っています。spawn を実行した後で、どのフィールドも使用されない限りは、再利用されたタスクを代わりに生成することができます。spawn を実行した後、親の処理が進む前に、タスクが実行され、破棄される可能性があるため、この制限は、const フィールドにも適用されます。

メモ: 同様のメソッド、task::recycle_as_continuation() は、タスクを子としてではなく継続として再利用します。

10.4.5 空のタスク

何もしないけれども、子が完了するまで待機するタスクが必要なことがあります。task.h ヘッダーは、この役割を担う empty_task クラスを定義します。定義は、次のとおりです。

```
// Task that does nothing. Useful for synchronization.
class empty_task: public task {
    /*override*/ task* execute() {
```

```
            return NULL;
        }
};
```

tbb/parallel_for.h の start_for::execute() メソッドで示されている empty_task が良い例です。ここでのコードは、継続渡しスタイルを使用します。これは、2 つの子タスクを作成し、子タスクの完了時に empty_task を継続として使用します。トップレベルのルーチン parallel_for (tbb/parallel_for.h 内) がルート上で待機します。

10.4.6 遅延コピー

データ構造は、別のスレッドがタスクをスチールしたときにのみコピーすると良いでしょう。例えば、tbb/parallel_reduce.h は、スレッドがスチールされたタスクを実行するときのみ提供した "ループ本体" オブジェクトをフォークする start_reduce::execute() メソッドを使用します。フォークでは、その後、スチールしたスレッドがタスクを終了するまでローカル実行させ、その結果を元のスレッドの結果と合わせます。fork/join ではわずかにオーバーヘッドが生じるため、スチールの発生時に行うのが唯一効果的です。

task::is_stolen_task メソッドは、スチールを検出する方法を提供します。実行中のタスク上で、通常はタスク自身がそれを呼び出します。タスクがスチールされた場合は非公式に true を返します。正式には、タスクを所有するスレッドが、スレッドの従属を所有するスレッドではない場合に true を返します。通常の fork-join タスクパターンの場合、従属を所有しているスレッドによってタスクが作成されるため、非公式な定義と正式な定義には同じ効果があります。例えば、従属は、親または親によって作成される継続です。

allocate_additional_child_of(t) メソッドが使用される場合は、この規則に例外が発生します。このメソッドは、*別の* タスク *t* が既に子を実行中であったとしても、*t* の子を生成するタスクによって使用されます。これは、兄弟が実行を開始する前に呼び出しを終了しなければならない allocate_child メソッドとは逆の動作です。allocate_additional_child_of の柔軟性により いくらかのオーバーヘッドがかかるため、この 2 つのメソッドは明確に区別できます。例えば、tbb/parallel_while.h では、allocate_additional_child_of メソッドは新しい兄弟を作成する子タスクの実行で使用されます。この場合、task::is_stolen_task は、*t* を実行中のスレッドによって子がスチールされない限り true になります。task::might_be_running_on_different_thread_than_dependent() という名前の方が正確ですが、長すぎるため、適切ではありません。

10.5 タスク・スケジューラーのまとめ

タスク・スケジューラーは、多くのフォークを持つ fork-join 並列化で最も効率良く動作します。タスクスチールが、スレッドをビジーな状態にしておくだけの十分な幅優先動作を行えるため、さらに作業をスチールする必要が出てくるまで、深さ優先の実行をスレッド自身で行うことができます。

タスク・スケジューラーは、速度向上を目的に設計されているため、単純ではありません。直接使用する必要がある場合は、`parallel_for` テンプレートや `parallel_reduce` テンプレートなどの、よりハイレベルなインターフェイスの背後に隠すと良いでしょう。いくつか覚えておくと良いことを次に紹介します。

- *allocation_method* が `task` クラスの割り当てメソッドのうちの 1 つである場合は、常に new(*allocation_method*) *T* を使用して、`task` を割り当てること。`task` のローカル・インスタンスまたはファイルスコープ・インスタンスは作成しないこと。
- `allocate_additional_child_of` を使用していない限りは、兄弟の実行が開始される前に、兄弟を割り当てること。
- 継続渡し、スケジューラー・バイパス、タスクの再利用を活用して、パフォーマンスを最大限に引き出すこと。
- タスクが終了し、再実行とマークされていない場合は、そのタスクは自動的に破棄される。また、その従属の参照カウントがデクリメントされ、ゼロに到達した場合は、その従属は自動的に生成される。

付録 A　タイムスライスのコスト

タイムスライスでは、物理スレッドよりも多くの論理スレッドが有効になります。各論理スレッドは、物理スレッドによって*タイムスライス*で実行されます。スレッドがタイムスライスよりも長時間実行される場合、次の順番がくるまでその物理スレッドは停止されます。この付録では、タイムスレッドによって発生するコストについて説明します。

最も明白なコストは、論理スレッド間の*コンテキストの切り替え*にかかる時間です。それぞれのコンテキスト・スイッチでは、プロセッサーで実行中だった前の論理スレッドのすべてのレジスターを確保し、次に実行する論理スレッドのレジスターをロードする必要があります。

また、それよりもコストが多いものに*キャッシュの冷却*があります。プロセッサーは、最近アクセスされたデータをキャッシュメモリーに保持します。キャッシュはメインメモリーに比べると、非常に処理が速いですが、容量は小さいものです。プロセッサーは、キャッシュメモリーを使い果たすと、キャッシュからアイテムを立ち退かせて、メインメモリーに戻します。この際、通常はキャッシュの中で最も古いものが選ばれます。(実際には、セット・アソシエーティブ・キャッシュはもう少し複雑です。) 論理スレッドがタイムスライスを取得し、データを初めて参照すると、このデータはキャッシュに置かれ、数百サイクルを消費します。データが頻繁に参照され、退去されない場合は、後続の参照がデータをキャッシュで得られるため、数サイクルが消費されるだけです。このようなデータを”キャッシュの中で熱い (ホット)” と言います。タイムスライスでは、スレッド A がそのタイムスライスを終了し、後続のスレッド B が同じ物理スレッドで実行されると、両方のスレッドでそのデータが必要でない限り、B は A のキャッシュでホットだったデータを立ち退かせる傾向があるため、このような操作は行われません。スレッド A が次のタイムスライスを取得すると、それぞれのキャッシュミスに数百サイクルを費やして、退去されたデータを再びロードします。さらに、スレッド A の次のタイムスライスは、異なるキャッシュを持つ異なる物理スレッド上にある可能性があります。

また、*ロック・プリエンプション*というコストもあります。この現象は、スレッドがリソース上でロックを取得し、そのタイムスライスが、ロックをリリースする前に時間切れになったときに発生する現象です。スレッドがロックをどれだけ短時間保持するように意図されていても、次のタイムスライスの順番になるまで保持されることになります。ロック上で待機している別のスレッドは、無駄にビジーウェイトになったり、あるいは、残りのタイムスライスを失うことになります。これは、前の

プリエンプトされたスレッドの実行が再開されるのを "数珠つなぎ" で待つため、*コンボイ現象* と呼ばれます。

付録 B　その他のスレッド化パッケージとの併用

インテル® スレッディング・ビルディング・ブロックは、他のスレッド化パッケージと併用することができます。他製品のスレッド化パッケージとともに、コンテナー、同期プリミティブ、アトミック操作を使用するために、特別な作業は必要ありません。ただし、並列化アルゴリズムやタスク・スケジューラーを使用する場合は、これらの機能を使用する各スレッドで、機能の使用中に有効なスレッド自身の task_scheduler_init オブジェクトを構築しなければならないため、追加の作業が必要です。

OpenMP で外部ループを並列化し、インテル® スレッディング・ビルディング・ブロックで内部ループを並列化する例を次に示します。

```
int M, N;

struct InnerBody {
    ...
};

void TBB_NestedInOpenMP() {
#pragma omp parallel
    {
        task_scheduler_init init;
#pragma omp for
        for( int i=0; i<M; ++j ) {
            parallel_for( blocked_range<int>(0,N,10), InnerBody(i) );
        }
    }
}
```

InnerBody の詳細は、ここでは省略します。重要なことは task_scheduler_init 宣言の配置です。#pragma omp parallel では、OpenMP はスレッドのチームを作成し、各スレッドはプラグマに関連付けられたブロック文を実行します。各スレッドは、自身の task_scheduler_init をブロック内に構築します。#pragma omp for は、コンパイラーが以前に作成されたスレッドチームを使用してループを並列で実行することを示します。これは、このプラグマはスレッドを作成しないため、対応する task_scheduler_init 宣言がないためです。

ここに、POSIX* スレッドを使用して記述された同じ例があります。

```
int M, N;

struct InnerBody {
    ...
};

void* OuterLoopIteration( void* args ) {
    task_scheduler_init init;
    int i = (int)args;
    parallel_for( blocked_range<int>(0,N,10), InnerBody(i) );
}

void TBB_NestedInPThreads() {
    std::vector<pthread_t> id( M );
    // Create thread for each outer loop iteration
    for( int i=0; i<M; ++i )
        pthread_create( &id[i], NULL, OuterLoopIteration, NULL );
    // Wait for outer loop threads to finish
    for( int i=0; i<M; ++i )
        pthread_join( &id[i], NULL );
}
```

参考文献

[1] “Memory Consistency & .NET”, Arch D. Robison, Dr. Dobb’s Journal, April 2003.

[2] A Formal Specification of Intel Itanium Processor Family Memory Ordering, Intel Corporation, October 2002.