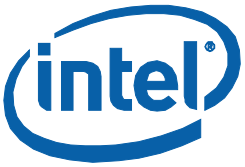

インテル® スレッディング・ビルディング・ブロック デザインパターン

デザインパターン

資料番号 323512-004JA

Web サイト: http://www.intel.com

著作権と商標について

最適化に関する注意事項

インテル® コンパイラー、関連ライブラリーおよび関連開発ツールには、インテル製マイクロプロセッサーおよび互換マイクロプロセッサーで利用可能な命令セット (SIMD 命令セットなど) 向けの最適化オプションが含まれているか、あるいはオプションを利用している可能性がありますが、両者では結果が異なります。また、インテル® コンパイラー用の特定のコンパイラー・オプション (インテル® マイクロアーキテクチャーに非固有のオプションを含む) は、インテル製マイクロプロセッサー向けに予約されています。これらのコンパイラー・オプションと関連する命令セットおよび特定のマイクロプロセッサーの詳細は、『インテル® コンパイラー・ユーザー・リファレンス・ガイド』の「コンパイラー・オプション」を参照してください。インテル® コンパイラー製品のライブラリー・ルーチンの多くは、互換マイクロプロセッサーよりもインテル製マイクロプロセッサーでより高度に最適化されます。インテル® コンパイラー製品のコンパイラーとライブラリーは、選択されたオプション、コード、およびその他の要因に基づいてインテル製マイクロプロセッサーおよび互換マイクロプロセッサー向けに最適化されますが、インテル製マイクロプロセッサーにおいてより優れたパフォーマンスが得られる傾向にあります。

インテル® コンパイラー、関連ライブラリーおよび関連開発ツールは、互換マイクロプロセッサー向けには、インテル製マイクロプロセッサー向けと同等レベルの最適化が行われない可能性があります。これには、インテル® ストリーミング SIMD 拡張命令 2 (インテル® SSE2)、インテル® ストリーミング SIMD 拡張命令 3 (インテル® SSE3)、ストリーミング SIMD 拡張命令 3 補足命令 (SSSE3) 命令セットに関連する最適化およびその他の最適化が含まれます。インテルでは、インテル製ではないマイクロプロセッサーに対して、最適化の提供、機能、効果を保証していません。本製品のマイクロプロセッサー固有の最適化は、インテル製マイクロプロセッサーでの使用を目的としています。

インテルでは、インテル® コンパイラーおよびライブラリーがインテル製マイクロプロセッサーおよび互換マイクロプロセッサーにおいて、優れたパフォーマンスを引き出すのに役立つ選択肢であると信じておりますが、お客様の要件に最適なコンパイラーを選択いただくよう、他のコンパイラーの評価を行うことを推奨しています。インテルでは、あらゆるコンパイラーやライブラリーで優れたパフォーマンスが引き出され、お客様のビジネスの成功のお役に立ちたいと願っております。お気づきの点がございましたら、お知らせください。

改訂 #20110307

改訂履歴

バージョン	バージョン情報	日付
1.04	最適化に関する注意事項を追加	2011 年 8 月 1 日
1.02	遅延初期化のサンプルを修正	2010 年 9 月 7 日
1.01	`enqueue_self` を `enqeue` に変更	2010 年 5 月 25 日
1.00	最初のバージョン	2010 年 4 月 4 日

目次

1 はじめに

このドキュメントは、いくつかの一般的な並列プログラミング・パターンと、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) を使用してそれらのパターンを実装する方法を説明した「クックブック」です。クックブックを読むだけで優れたシェフになることはできませんが、役立つ数々のレシピを知ることはできます。

ほとんどのクックブックと同様に、このドキュメントでは読者が基本的なツールの使用方法を知っていると仮定しています。基本的なツールの使用方法については、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) のチュートリアルを参照してください。このドキュメントでは、どのツールをいつ使用するかを説明します。

デザインパターンの説明は、暗記するのではなく、理解することが必要です。各パターンの説明は、次の形式で行っています。

- **問題** – 解決する問題を説明します。
- **コンテキスト** – 問題が発生するコンテキストを説明します。
- **影響** – パターンの使用による影響の考察です。
- **ソリューション** – パターンの実装方法を説明します。
- **サンプル** – 実装例を示します。

バリエーションとサンプルは状況に合わせて紹介します。コードサンプルは重要なポイントを強調するために用いられているものであり、完全なコードではありません。非 const メソッドの明示的な const オーバーロードは省略しています。

用語とサンプルの多くは、Eun-Gyu と Marc Snir によって作成された Web ページとカリフォルニア大学バークレー校の parallel patterns wiki の内容を参考にしています。「全般的な参考文献」セクションのリンクを参照してください。

コードを簡潔にするため、いくつかのコードサンプルでは C++0x ラムダ式を使用しています。ラムダ式を等価な C++98 コードで表現することは難しくありませんが、少し冗長になることがあります。インテル® コンパイラーでラムダ式を有効にする方法や手動で変換する方法については、インテル® TBB のチュートリアルの「ラムダ式」セクションを参照してください。

2 凝集化

問題

並列化の粒度が細かすぎて、並列スケジューリングや通信のオーバーヘッドが大きすぎる。

コンテキスト

多くのアルゴリズムでは、タスクあたりの命令数が少ない、非常に細かな粒度での並列化を行うことができます。しかし、スレッド間の同期には通常、桁違いに大きなサイクル数が必要になります。例えば、2 つの配列の要素単位の加算は完全に並列に行うことができますが、スカラー加算が別のタスクとしてスケジュールされた場合、ほとんどの時間は加算ではなく同期に費やされるでしょう。

影響

- 個々の計算は並列に行うことが可能ですが、小さいものです。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) の実用的な処理では、「小さい」とは 10,000 サイクル未満を意味します。
- 並列化はパフォーマンス向上が目的であり、セマンティクス上の理由から必要な訳ではありません。

ソリューション

計算のグループをブロックにします。ブロック内の計算をシリアルに評価します。

ブロックサイズは (並列化のオーバーヘッドを考慮してもメリットがある) 十分な大きさにしてください。ブロックサイズが大きすぎると、ブロック数が非常に少なくなり、プロセッサー間で平等に処理を分散できなくなるため、並列化やロード・バランシングが制限されることがあります。

ブロックサイズは通常、次の 2 点を考慮して決定します。

- ブロック間の同期を最小限にする。
- ブロック間のキャッシュ・トラフィックを最小限にする。

計算が完全に独立していて、ブロックも独立している場合、キャッシュ・トラフィックのみを考慮してください。

ループが「小さい」場合 (10,000 クロックサイクル未満の場合)、最適な集合が単一ブロックになるため、ループを並列化するのは非実用的です。

サンプル

*範囲*引数が指定できるインテル® TBB のループ・テンプレート (`tbb::parallel_for` など) は、自動凝集化をサポートしています。

凝集化を行うときは、キャッシュの影響を考慮してください。可能であれば、キャッシュラインがグループ間をまたがないようにしてください。

キャッシュライン境界は内部のレートに影響します。例えば、計算で 2D グリッドが形成され、最も近いポイントとのみ通信する場合、ブロックごとの計算は (ブロックのエリアで) 2 次的に増加しますが、ブロック間の通信は (ブロックの周囲で) 直線的に増加します。図 1 は、8×8 グリッドを凝集化する 4 つの異なる方法を示しています。このような解析を行う場合、キャッシュライン単位で転送される情報を考慮するようにしてください。ここでは、ブロックが論理グリッドに関して正方形ではなく、キャッシュラインのグリッドに関して正方形の場合に、周囲が最小になります。

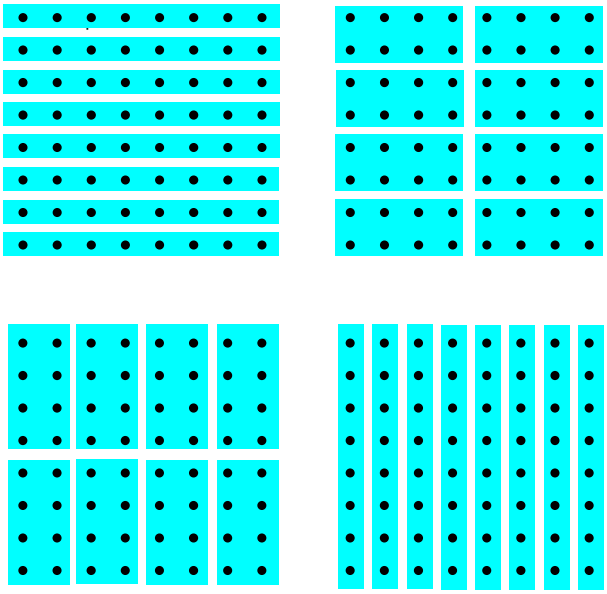

図 1: 8×8 グリッドの 4 つの異なる凝集化

ベクトル化についても考慮してください。長い連続したデータのサブセットを含むブロックには、ベクトル化が有効です。

再帰計算では、ほとんどの処理は木構造の葉ノードで行われるため、ソリューションは図 2 に示すようにグループを部分木として扱うことです。

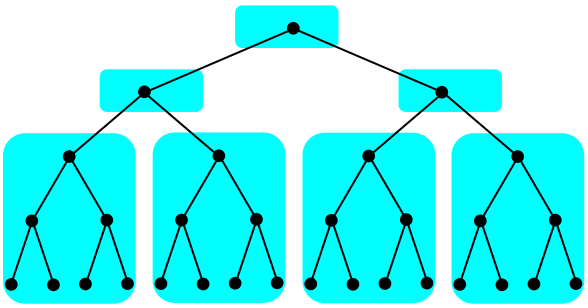

図 2: 再帰計算の凝集化

多くの場合、このような凝集化は、あるしきい値に達すると再帰的に続けて行われます。例えば、再帰ソートは、特定のしきい値を超えた場合にのみ部分問題を並列で解きます。

参考文献

用語 "agglomeration" は、Ian Foster が著書 *Designing and Building Parallel Programs* <http://www.mcs.anl.gov/~itf/dbpp> で使用したものです。Agglomeration は、4 ステップの「PCAM」設計法の一部にも含まれています。

1. **P**artitioning (分割) - プログラムをできるだけ小さなタスクに分割します。
2. **C**ommunication (通信) – タスク間で必要な通信を特定します。インテル® TBB を使用する場合、通常はキャッシュラインの転送を使用します。これは自動で行われますが、タスク間で起きていることを理解すると、凝集化ステップを進めやすくなります。
3. **A**gglomeration (凝集化) – タスクを組み合わせてより大きなタスクにします。彼の書籍には、さまざまな考察の広範なリストが含まれています。
4. **M**apping (マッピング) – タスクをプロセッサーにマップします。インテル® TBB では、タスク・スケジューラーがこのステップを行います。

3 要素単位

問題

データセットの各項目で同様の独立した計算が開始され、すべて完了するまで待機する。

コンテキスト

多くのシリアル・アルゴリズムは、項目のセットをスイープして各項目の独立した計算を行います。ただし、ある種のサマリー情報が収集される場合は、代わりにリダクション・パターンを使用します。

影響

計算中に情報が伝えられません (またはマージされません)。

ソリューション

項目数があらかじめわかっている場合は、`tbb::parallel_for` を使用します。わかっていない場合は、`tbb::parallel_do` を使用します。

個々の計算がスケジューラーのオーバーヘッドよりも相対的に小さい場合は、凝集化を使用します。

同じデータに対するリダクションがパターンに続く場合は、リダクションの一部として要素単位の操作を行う (2 つのパターンの組み合わせを 2 回のスイープではなく 1 回のスイープで行う) ことを検討してみてください。メモリー階層間のトラフィックが減ることでパフォーマンスが向上する可能性があります。

サンプル

畳み込みは信号処理でよく使用されます。フィルター c と信号 x の畳み込みは次のように計算されます。

$$y_i = \sum_j c_j x_{i-j}$$

この計算のシリアルコードは次のようになります。

```
// c[0..clen-1] と x[1-clen..xlen-1] が定義されていると仮定します
for( int i=0; i<xlen+clen-1; ++i ) {
    float tmp = 0;
    for( int j=0; j<clen; ++j )
        tmp += c[j]*x[i-j];
    y[i] = tmp;
}
```

簡潔に表記するため、x は *x*[*k*] (*k*<0 または *k*≥*xlen* の場合にゼロを返す) のようにゼロでパディングされる配列のポインターであると仮定しています。

各反復は前の反復に依存するため、内部ループは要素単位パターンに自然に適合しませんが、外部ループは要素単位パターンに適合します。上記のコードは、tbb::parallel_for を使用して次のように表現できます。

```
tbb::parallel_for( 0, xlen+clen-1, [=]( int i ) {
    float tmp = 0;
    for( int j=0; j<clen; ++j )
        tmp += c[j]*x[i-j];
    y[i] = tmp;
});
```

tbb::parallel_for は、その実装で暗黙的に tbb::auto_partitioner を使用して自動で凝集化を行います。明示的に凝集化を行う理由がある場合は、明示的な範囲引数を指定できる tbb::parallel_for のオーバーロードを使用します。オーバーロードを使用するように変更したサンプルを次に示します。

```
tbb::parallel_for(
    tbb::blocked_range<int>(0,xlen+clen-1,1000),
    [=]( tbb::blocked_range<int> r ) {
        int end = r.end();
        for( int i=r.begin(); i!=end; ++i ) {
            float tmp = 0;
            for( int j=0; j<clen; ++j )
                tmp += c[j]*x[i-j];
            y[i] = tmp;
        }
    }
);
```

4 奇偶通信

問題

データの操作を完全に独立して行うことはできないが、サブセットのすべての操作を並列で実行できる 2 つのサブセットに分割することはできる。

コンテキスト

偏微分方程式のソルバーは、このパターンに従うように修正できます。例えば、最も近い地点とのみ通信する 2D グリッドの場合、グリッドをチェス盤とみなして、赤のマスと黒のマスを交互に更新することができます。

または、互い違いに配列されたグリッド (leap frog、蛙飛び) に、このパターンに適合する時間領域差分 (FDTD) ソルバーを使用します。`examples/parallel_for/seismic/` のコードでは、このようなソルバーを使用しています。

影響

- 項目間の依存関係により 2 部グラフが形成されます。

ソリューション

1 つのサブセットを更新した後、もう 1 つのサブセットを (交互に) 更新します。各サブセットに要素単位パターンを適用します。

サンプル

`examples/parallel_for/seismic` のサンプルは、原理を示しています。ここで、2 つの物理フィールド *velocity* と *stress* は互いに依存しており、同時に更新することができません。ただし、*velocity* の計算は *stress* の値が固定である限り行うことができ、逆も同様です。このため、コードは *velocity* と *stress* フィールドを交互に更新します。更新はそれぞれ、`tbb::parallel_for` を使用して行われます。

参考文献

"Odd-Even Communication Group" <http://www.cs.uiuc.edu/homes/snir/PPP/patterns/oddeven.pdf> Eun-Gyu Kim と Marc Snir によるパターンの説明。

5 ウェーブフロント

問題

先行する項目 (predecessor) の計算結果を使用してデータセットの項目の計算を行う。詳細は、「参考文献」を参照してください。

コンテキスト

計算間の依存関係により閉路のないグラフが形成されます。

影響

- 項目間の依存関係の制約により閉路のないグラフが形成されます。
- グラフの直前の先行する項目の数はあらかじめわかっているか、最後の先行する項目が完了する前に決定されます。

ソリューション

項目の処理に `tbb::parallel_do` を使用して、トポロジカル・ソートを並列で行います。各項目にアトミックカウンターを関連付けます。各カウンターを先行する項目の数に初期化します。tbb::parallel_do を起動して先行する項目のない項目 (カウントがゼロ) を処理します。項目が処理されたら、それに続く項目 (successor) のカウンターをデクリメントします。続く項目のカウンターがゼロになったら、「feeder」により続く項目を `tbb::parallel_do` に追加します。

項目に先行する項目の数があらかじめ決定できない場合、「既知の先行する項目の数 (know number of predecessors)」情報を追加の先行する項目として扱います。先行する項目の数が分かったら、この概念的な先行する項目を完了として扱います。

個々の項目をカウントするオーバーヘッドが大きい場合、複数の項目を凝集化して、そのブロックに対してウェーブフロントを行います。

サンプル

最長共通部分列アルゴリズムのシリアルカーネルを次に示します。パラメーターは文字列 x と y (長さは `xlen` と `ylen`) です。

```
int F[MAX_LEN+1][MAX_LEN+1];

void SerialLCS( const char* x, size_t xlen, const char* y, size_t ylen )
{
    for( size_t i=1; i<=xlen; ++i )
        for( size_t j=1; j<=ylen; ++j )
            F[i][j] = x[i-1]==y[j-1] ? F[i-1][j-1]+1 :
                                       max(F[i][j-1],F[i-1][j]);
}
```

カーネルは、x[0..i-1] および y[0..j-1] で共有される最長共通部分列の長さに `F[i][j]` をセットします。F[0][0..ylen] および F[0..xlen][0] はすでにゼロに初期化されていると仮定します。

図 3 は、`F[i][j]` 計算のデータの依存関係を示しています。

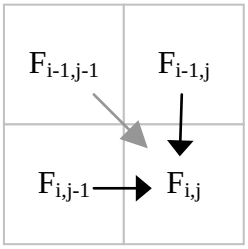

図 3: 最長共通部分文字列計算のデータの依存関係

次の図 4 に示すように、灰色の対角線の依存関係はほかの依存関係の過渡的なクロージャーです。このため、並列化目的では、この依存関係を無視できます。

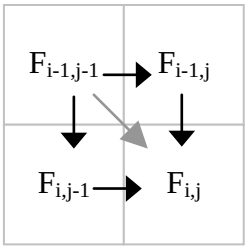

図 4: 対角線の依存関係は冗長

アトミックカウントは依存関係を考慮するコストがかかるため、冗長な依存関係を考慮しないようにすることは一般によい方法です。

もう 1 つ考慮すべき点は粒度です。F[i][j] 要素計算をそれぞれ別々にスケジュールすることはコストがかかりすぎます。よいソリューションは、複数の要素を連続するブロックに擬集化して、ブロックのコンテンツを連続して処理することです。ブロックの依存関係のパターンは同じですが、ブロックはスケーリングされます。このため、スケジューリングのオーバーヘッドを考慮してもメリットがあります。

並列コードを次に示します。各ブロックは `N×N` の要素で構成されます。各ブロックにはアトミックカウンターが関連付けられています。配列 `Count` は、簡単に検査できるように、これらのカウン

ターを構成します。コードはカウンターを初期化した後、parallel_do を使用して、ウェーブフロントを回転します (predecessor がないため原点のブロックから開始)。

```
const int N = 64;
tbb::atomic<char> Count[MAX_LEN/N+1][MAX_LEN/N+1];

void ParallelLCS( const char* x, size_t xlen, const char* y, size_t ylen )
{
    // ブロックの predecessor カウントを初期化します
    size_t m = (xlen+N-1)/N;
    size_t n = (ylen+N-1)/N;
    for( int i=0; i<m; ++i )
        for( int j=0; j<n; ++j )
            Count[i][j] = (i>0)+(j>0);

    // ウェーブフロントを原点から回転します
    typedef pair<size_t,size_t> block;
    block origin(0,0);
    tbb::parallel_do( &origin, &origin+1,
        [=]( const block& b, tbb::parallel_do_feeder<block>& feeder ) {

            // ブロックの境界を抽出します
            size_t bi = b.first;
            size_t bj = b.second;
            size_t xl = N*bi+1;
            size_t xu = min(xl+N,xlen+1);
            size_t yl = N*bj+1;
            size_t yu = min(yl+N,ylen+1);

            // ブロックを処理します
            for( size_t i=xl; i<xu; ++i )
                for( size_t j=yl; j<yu; ++j )
                    F[i][j] = x[i-1]==y[j-1] ? F[i-1][j-1]+1 :
                                               max(F[i][j-1],F[i-1][j]);

            // successor を考慮します
            if( bj+1<n && --Count[bi][bj+1]==0 )
                feeder.add( block(bi,bj+1) );
            if( bi+1<m && --Count[bi+1][bj]==0 )
                feeder.add( block(bi+1,bj) );        }
    );
}
```

規則的な構造はウェーブフロントの実装を単純化しますが、必須ではありません。examples/parallel_do/parallel_preorder の並列前順走査は、predecessor が走査された後にノードが走査されるという制約に従って、ウェーブフロント・パターンをグラフの各ノードの走査に並列で適用します。このサンプルで、グラフ中の各ノードは predecessor のカウントを格納します。

参考文献

最長共通部分列のサンプルは、Eun-Gyu Kim と Marc Snir による “Wavefront Pattern” <http://www.cs.illinois.edu/homes/snir/PPP/patterns/wavefront.pdf> を参考に作成しました。

6 リダクション

問題

データセットにまたがって結合則を満たすリダクション操作を行う。

コンテキスト

多くのシリアル・アルゴリズムは、項目のセットをスイープしてサマリー情報を収集します。

影響

サマリーはデータセットの結合則を満たす操作として表現されます。または、少なくとも再結合が重要でないほど結合則を満たしたものになります。

ソリューション

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) には 2 つのソリューションが含まれています。どちらのソリューションを使用するかは、次の考察に依存します。

- 操作は可換で結合則を満たす操作か?
- リダクション型のインスタンスの構築と破棄にコストがかかるか? 例えば、浮動小数点数の構築にはコストはかかりませんが、浮動小数点数の疎行列の構築にはコストが非常にかかるかもしれません。

オブジェクトを構築するコストがあまりかからない場合は、`tbb::parallel_reduce` を使用します。これは、リダクション操作が可換でない場合も動作します。インテル® TBB のチュートリアルでは、基本的なリダクションで `tbb::parallel_reduce` を使用する方法を説明しています。

リダクション操作が可換で型のインスタンスにコストがかかる場合、`tbb::parallel_for` と `tbb::combinable` を使用します。

操作が厳密には結合則を満たしていないが厳密な決定性による結果が必要な場合は、再帰的なリダクションを使用して、`tbb::parallel_invoke` で並列化します。

サンプル

次のサンプルは、さまざまなソリューションといくつかのトレードオフを示しています。

最初のサンプルは、`tbb::parallel_reduce` を使用して型 T のシーケンスに a+ リダクションを行っています。シーケンスは、半開区間 [first,last) で定義されます。

```
T AssocReduce( const T* first, const T* last, T identity ) {
    return tbb::parallel_reduce(
         // リダクションのインデックス範囲を指定します
        tbb::blocked_range<const T*>(first,last),
         // 単位元要素
        identity,
         // 部分範囲と部分和をまとめます
        [&]( tbb::blocked_range<const T*> r, T partial_sum )->float {
            return std::accumulate( r.begin(), r.end(), partial_sum );
        },
         // 2 つの部分和をまとめます
        std::plus<T>()
    );
}
```

この形式の parallel_reduce の第 3 および第 4 引数は、擬集化パターンの形式で構築されます。リダクションの前に要素単位動作を行わなければならない場合、第 3 引数 (部分範囲のリダクション) に動作を追加すると、参照の局所性が向上し、パフォーマンスが改善されます。

2 つめの例は、+ が T で可換であると仮定しています。T オブジェクトの構築にコストがかかる場合にはよいソリューションです。

```
T CombineReduce( const T* first, const T* last, T identity ) {
    tbb::combinable<T> sum(identity);
    tbb::parallel_for(
        tbb::blocked_range<const T*>(first,last),
        [&]( tbb::blocked_range<const T*> r ) {
            sum.local() += std::accumulate(r.begin(), r.end(), identity);
        }
    );
    return sum.combine( []( const T& x, const T& y ) {return x+y;} );
}
```

最終的な結果を生成するために部分的な結果を使用することが望ましい場合があります。例えば、部分的な結果がリストの場合、リストを結合して最終的な結果を生成することができます。その場合、combinable の代わりにクラス tbb::enumerable_thread_specific を使用します。第 7 章のサンプル ParallelFindCollisions は、そのテクニックを示しています。

浮動小数点の加算と乗算はほぼ結合則を満たしています。再結合を行うと、丸めの影響により結果が変更されることがあります。次に示すテクニックは、項目を非決定的に再結合します。全く結合則を満たさない操作に対して完全に決定性のある並列リダクションを行うには、決定性のある再結合を使用する必要があります。次のコードは、型 T の値のシーケンスに a + リダクションを行うテンプレートの形式で、この処理を示しています。

```
template<typename T>
T RepeatableReduce( const T* first, const T* last, T identity ) {
    if( last-first<=1000 ) {
        // シリアル・リダクションを使用します
        return std::accumulate( first, last, identity );
    } else {
        // 並列の分割統治リダクションを行います
        const T* mid = first+(last-first)/2;
        T left, right;
        tbb::parallel_invoke(
            [&]{left=RepeatableReduce(first,mid,identity);},
            [&]{right=RepeatableReduce(mid,last,identity);}
        );
        return left+right;
    }
}
```

外側の if-else は、再帰計算の擬集化パターンのインスタンスです。リダクション・グラフは厳密な二分木ではありませんが、完全な決定性があります。このため、すべてのスレッドで浮動小数点の丸めが同一であれば、指定された入力シーケンスで結果は常に同じになります。

最後のサンプルは、通常はリダクションとは見なされない問題をリダクションとして見なすことで並列化する方法を示しています。この問題は、データセットにまたがって計算の浮動小数点例外フラグを取得しています。シリアルコードは次のようになります。

```
    feclearexcept(FE_ALL_EXCEPT);
    for( int i=0; i<N; ++i )
        C[i]=A[i]*B[i];
    int flags = fetestexcept(FE_ALL_EXCEPT);
    if (flags & FE_DIVBYZERO) ...;
    if (flags & FE_OVERFLOW) ...;
    ...
```

コードは、ループのチャンクを別々に計算して各チャンクの浮動小数点フラグをマージすることで並列化できます。この処理を tbb:parallel_reduce で行うには、次に示すように、最初に「本体」の型を定義します。

```
struct ComputeChunk {
    int flags;           // これまでに見つかった浮動小数点例外を保持します
    void reset_fpe() {
        flags=0;
        feclearexcept(FE_ALL_EXCEPT);
    }
    ComputeChunk () {
        reset_fpe();
    }
    // 範囲を部分範囲に分割するとき parallel_reduce に呼び出される "分割コンストラクター"
    ComputeChunk ( const ComputeChunk&, tbb::split ) {
        reset_fpe();
    }
```

```
    // チャンクを操作し、浮動小数点例外の状態を収集して flags メンバーに設定します
    void operator()( tbb::blocked_range<int> r ) {
        int end=r.end();
        for( int i=r.begin(); i!=end; ++i )
            C[i] = A[i]/B[i];
    // ここで = ではなく |= を使用することが重要です
    // そうしないと同一スレッドにおいて先に見つかっている例外が失われます
        flags |= fetestexcept(FE_ALL_EXCEPT);
    }
    // 2 つの部分範囲の結果を結合するとき parallel_reduce に呼び出されます
    void join( Body& other ) {
        flags |= other.flags;
    }
};
```

その後、次のように起動します。

```
    // cc の構築によって FP 例外状態が暗黙的にリセットされます
    ComputeChunk cc;
    tbb::parallel_reduce( tbb::blocked_range<int>(0,N), cc );
    if (cc.flags & FE_DIVBYZERO) ...;
    if (cc.flags & FE_OVERFLOW) ...;
    ...
```

7 分割統治

問題

分割統治アルゴリズムを並列化する。

コンテキスト

分割統治はシリアル・アルゴリズムで広く使用されています。一般的な例は、クイックソートとマージソートです。

影響

- 問題は別々に解くことができる複数の部分問題に変形することができます。
- 問題の分割やソリューションのマージのコストは、部分問題を解くコストと比較して相対的に低くなります。

ソリューション

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) で分割統治を実装する方法はいくつかあります。最適な選択肢は状況に依存します。

- 分割で同じ数の部分問題が常に生成される場合は、再帰と `tbb::parallel_invoke` を使用します。
- 部分問題の数が異なる場合は、再帰と `tbb::task_group` を使用します。
- 効率とスケーラビリティーが重要な場合は、`tbb::task` と継続渡し形式を使用します。

サンプル

クイックソートは古典的な分割統治アルゴリズムで、ソート問題を 2 つの部分ソートに分割します。単純なシリアルバージョンは次のようになります。[1]

[1] 製品品質のクイックソートの実装では通常、小さな部分ソートには、より高度なピボット選択、再帰の代わりに明示的なスタック、ほかのソート・アルゴリズムを使用します。ここでは、並列パターンの解説に焦点を当てるため単純なアルゴリズムを使用しています。

```
void SerialQuicksort( T* begin, T* end ) {
    if( end-begin>1  ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        SerialQuicksort( begin, mid-1 );
        SerialQuicksort( mid, end );
    }
}
```

部分ソートの数は 2 つに固定されているため、tbb::parallel_invoke を使用して単純な方法で部分ソートを並列化します。並列コードを次に示します。

```
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>1 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                              [=]{ParallelQuicksort( mid, end );} );
    }
}
```

部分ソートが小さくなりすぎるとシリアル実行のほうが効率がよくなるため、次のバージョン (変更点を青で表示) では、以前のシリアルコードを使用して 500 未満の要素のソートを行っています。

```
void ParallelQuicksort( T* begin, T* end ) {
    if( end-begin>=500 ) {
        using namespace std;
        T* mid = partition( begin+1, end, bind2nd(less<T>(),*begin) );
        swap( *begin, mid[-1] );
        tbb::parallel_invoke( [=]{ParallelQuicksort( begin, mid-1 );},
                              [=]{ParallelQuicksort( mid, end );} );
    } else {
        SerialQuicksort( begin, end );
    }
}
```

変更は擬集化パターンのインスタンスです。

次のサンプルでは、部分問題の数が異なる問題を扱います。問題は、木構造を含んでいて、2 種類のノードがあります。

- 葉ノードは個々のパーツを表します。
- 内部ノードはパーツのグループを表します。

問題は、ターゲットのノードと衝突するノードをすべて検索します。次のコードは、ツリーを移動するシリアル・ソリューションを示しています。ここでは、Hits に Target と衝突したノードをすべて記録します。

```
std::list<Node*> Hits;
Node* Target;

void SerialFindCollisions( Node& x ) {
    if( x.is_leaf() ) {
        if( x.collides_with( *Target ) )
            Hits.push_back(&x);
    } else {
        for( Node::const_iterator y=x.begin(); y!=x.end(); ++y )
            SerialFindCollisions(*y);
    }
}
```

並列バージョンを次に示します。

```
typedef tbb::enumerable_thread_specific<std::list<Node*> > LocalList;
LocalList LocalHits;

Node* Target;     // ターゲットノード

void ParallelWalk( Node& x ) {
    if( x.is_leaf() ) {
        if( x.collides_with( *Target ) )
            LocalHits.local().push_back(&x);
    } else {
        // x の子 y を並列で再帰的に処理します
        tbb::task_group g;
        for( Node::const_iterator y=x.begin(); y!=x.end(); ++y )
            g.run( [=]{ParallelWalk(*y);} );
        // 再帰呼び出しが完了するまで待機します
        g.wait();
    }
}

void ParallelFindCollisions( Node& x ) {
    ParallelWalk(x);
    for(LocalList::iterator i=LocalHits.begin(); i!=LocalHits.end(); ++i)
        Hits.splice( Hits.end(), *i );
}
```

再帰移動は、並列で再帰呼び出しを行うためにクラス task_group を使用して並列化されます。

並列性の導入による、別の重要な変更があります。Hits を同時に更新することは安全ではないため、並列移動は変数 LocalHits を使用して結果を蓄積します。この変数は enumerable_thread_specific 型なので、各スレッドは個々の結果を蓄積できます。移動が完了した後、結果が Hits に結合されます。

結果はオリジナルのシリアルコードと同じ順序に*なりません*。

並列化のオーバーヘッドが大きい場合は、擬集化パターンを使用します。例えば、特定のしきい値未満の部分木にシリアル移動を使用します。

8 GUI スレッド

問題

ユーザー・インターフェイス・スレッドはユーザーの要求に反応する必要がある。長い計算の実行中でも反応しなければならない。

コンテキスト

グラフィカル・ユーザー・インターフェイスには、多くの場合、ユーザーの操作を処理する専用のスレッド (GUI スレッド) があります。アプリケーションが長い計算を実行している間であっても、スレッドはユーザーの要求に反応する必要があります。例えば、ユーザーが長く実行している計算を停止するために「キャンセル」ボタンを押すことがあります。GUI スレッドがこの計算に関係している場合、ユーザーの要求に反応することができません。

影響

- GUI スレッドはイベントループをサービスします。
- GUI スレッドはほかのスレッドに作業を渡し、その作業の完了を待機しないようにする必要があります。
- GUI スレッドはイベントループに反応しなければならず、渡された作業を行うことに専念してはなりません。

関連項目

ノンプリエンプティブな優先度

ローカル・シリアライザー

ソリューション

GUI スレッドは、メソッド `task::enqueue` を使用してタスクに作業を渡します。作業が完了すると、タスクは作業が完了したことを知らせるイベントを GUI スレッドにポストします。メソッド enqueue のセマンティクスにより、タスクは呼び出しスレッドとは異なるワーカースレッドで実行されます。このメソッドは、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) 3.0 の新機能です。

図 5 は、通信パスを図で示したものです。黒の項目は GUI スレッドで実行され、青の項目は別のスレッドで実行されます。

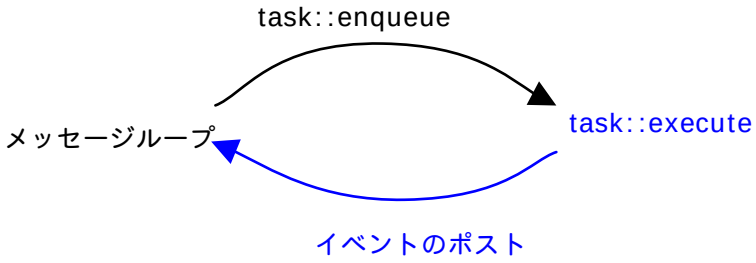

図 5: GUI スレッドパターン

サンプル

このサンプルは Microsoft* Windows* オペレーティング・システム向けですが、同様の原理はイベントループを使用する任意の GUI に適用できます。イベントごとに、GUI スレッドはユーザー定義関数 WndProc を呼び出してイベントを処理します。重要な部分は太字で表記しています。

```
// 作業完了時にキューに入れられたタスクからポストされるイベント
const UINT WM_POP_FOO = WM_USER+0;
// キューに入れられたタスクから GUI スレッドに結果を送信するためのキュー
tbb::concurrent_queue<Foo> ResultQueue;
// GUI スレッドの最新の計算結果のプライベート・コピー
Foo CurrentResult;

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM
lParam) {
    switch(msg) {
        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDM_LONGRUNNINGWORK:
                    // ユーザーが要求した長い計算を 別のスレッドに依頼します
                    LaunchLongRunningWork(hWnd);
                    break;
                case IDM_EXIT:
                    DestroyWindow(hWnd);
                    break;
                default:
                    return DefWindowProc(hWnd, msg, wParam, lParam);
            }
            break;
        case WM_POP_FOO:
            // ResultQueue に処理すべき別の結果があります
            ResultQueue.try_pop(CurrentResult);
            // 最新の結果でウィンドウを更新します
            RedrawWindow( hWnd, NULL, NULL, RDW_ERASE|RDW_INVALIDATE );
```

```
            break;
        case WM_PAINT:
            CurrentResult を使用してウィンドウを再描画します
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc( hWnd, msg, wParam, lParam );
    }
    return 0;
}
```

GUI スレッドは長い計算を次のように処理します。

1. GUI スレッドは LongRunningWork を呼び出して作業をワーカースレッドに依頼します。
2. GUI スレッドはイベントループのサービスを続けます。ウィンドウの再描画が必要な場合、Foo が最後に処理した最新の値である CurrentResult を使用します。

ワーカーが長い計算を終了すると、結果を ResultQueue にプッシュして、GUI スレッドに WM_POP_FOO メッセージを送信します。

3. GUI スレッドは ResultQueue から CurrentResult に項目をポップして WM_POP_FOO メッセージをサービスします。ResultQueue の各項目に WM_POP_FOO メッセージが 1 つあるため、try_pop は常に成功します。

ルーチン LaunchLongRunningWork はルートタスクを作成し、メソッド task::enqueueue を使用してタスクを起動します。待機している後続のタスクがないため、このタスクはルートタスクです。

```
class LongTask: public tbb::task {
    HWND hWnd;
    tbb::task* execute() {
        Do long computation
        Foo x = result of long computation
        ResultQueue.push( x );
        // 結果が利用可能であることを GUI スレッドに通知します
        PostMessage(hWnd,WM_POP_FOO,0,0);
        return NULL;
    }
public:
    LongTask( HWND hWnd_ ) : hWnd(hWnd_) {}
};

void LaunchLongRunningWork( HWND hWnd ) {
    LongTask* t = new( tbb::task::allocate_root() ) LongTask(hWnd);
    tbb::task::enqueue(*t);
}
```

メソッド task::spawn ではなくメソッド task::enqueue を使用する必要があります。その理由は、メソッド enqueue は、タスクを明示的に待っているスレッドがない場合でも、リソース的に問題がなければタスクを実行するためです。対照的に、メソッド spawn は、明示的に待たれるまでタスクの実行を延期します。

サンプルでは、concurrent_queue を使用してワーカースレッドから GUI スレッドに結果を返しています。サンプルでは最新の結果のみ重要なので、別の方法としてミューテックスで保護されている共有変数を使用する方法もあります。ただし、この方法では GUI スレッドがミューテックスをロックしている間はワーカーがブロックされます。逆も同様です。このため、concurrent_queue を使用するほうが、単純かつ優れたソリューションです。

2 つの長い計算を実行したときに、最初の計算が 2 つめの計算の後に完了する可能性があります。最後に要求した計算の結果を表示することが重要なのであれば、要求にシリアル番号を割り当て、計算と関連付けます。GUI スレッドは ResultQueue をポップして一時変数に入れ、シリアル番号を確認して、シリアル番号が進む場合のみ CurrentResult を更新します。

優先度の実装方法については、「ノンプリエンプティブな優先度」を参照してください。特定のタスクを強制的に連続した順序にする方法については、「ローカル・シリアライザー」を参照してください。

9 ノンプリエンプティブな優先度

問題

優先度に基づいて次に処理する作業を選択する。

コンテキスト

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) のスケジューラーは、スケーラビリティーに基づく規則を使用してタスクを選択します。この規則は、タスクがスポーンまたはキューに入れられた順序に基づいており、タスクの内容は考慮しません。場合によっては、優先度の関係に基づいて作業を選択するほうがよいこともあります。

影響

- 複数の作業項目を指定した場合、(デフォルトのインテル® TBB の規則では*ない*) 次に処理する項目についての規則があります。
- プリエンプティブな優先度は必要ありません。優先度の高い項目が現れた場合、実行中の優先度の低い項目を直ちに停止する必要はありません。プリエンプティブな優先度が必要な場合、ノンプリエンプティブなタスク処理は不適切です。代わりにスレッドを使用してください。

ソリューション

作業を共有作業パイルに入れます。特定の作業からタスクを切り離します。その結果、タスク実行はパイルから実際の仕事を選択します。

サンプル

次のサンプルは、3 つの優先度を実装します。ユーザー・インターフェイスとトップレベルの実装が続きます。

```
enum Priority {
    P_High,
    P_Medium,
    P_Low
};

template<typename Func>
void EnqueueWork( Priority p, Func f ) {
    WorkItem* item = new ConcreteWorkItem<Func>( p, f );
    ReadyPile.add(item);
}
```

呼び出し元はルーチン EnqueueWork に優先度 *p* とファンクター *f* を提供します。ファンクターはラムダ式によるものかもしれません。EnqueueWork は *f* を WorkItem としてパッケージし、グローバル・オブジェクト ReadyPile に追加します。

クラス WorkItem は、不明な型のファンクターを実行するための統一されたインターフェイスを提供します。

```
// 優先度を付ける作業の抽象基本クラス
class WorkItem {
public:
    WorkItem( Priority p ) : priority(p) {}
    // 派生クラスは実際の作業を定義します
    virtual void run() = 0;
    const Priority priority;
};

template<typename Func>
class ConcreteWorkItem: public WorkItem {
    Func f;
    /* オーバーライド */
   void run() {
        f();
        delete this;
    }
public:
    ConcreteWorkItem( Priority p, const Func& f_ ) :
        WorkItem(p), f(f_)
    {}
};
```

クラス ReadyPile はコアパターンを含みます。作業のコレクションを保持して、コレクションから作業を選択するタスクを送ります。

```
class ReadyPileType {
    // 各優先度レベルごとに 1 つのキュー
    tbb::concurrent_queue<WorkItem*> level[P_Low+1];
public:
    void add( WorkItem* item ) {
        level[item->priority].push(item);
        tbb::task::enqueue(*new(tbb::task::allocate_root()) RunWorkItem);
    }
    void runNextWorkItem() {
        // 項目の優先度順にキューをスキャンします
        WorkItem* item=NULL;
        for( int i=P_High; i<=P_Low; ++i )
            if( level[i].try_pop(item) )
                break;
        assert(item);
        item->run();
    }
```

```
};

ReadyPileType ReadyPile;
```

add(item) によってキューに入れられたタスクは、必ずしもその項目を実行するわけでは*ありません*。タスクは runNextWorkItem() を実行し、より優先度の高い項目が見つかることがあります。各項目につき 1 つのタスクがありますが、マッピングにより、タスクがいつ作成されるかではなく、タスクがいつ実際に実行されるかを指定できます。

クラス RunWorkItem の詳細は次のとおりです。

```
class RunWorkItem: public tbb::task {
    /* オーバーライド */
   tbb::task* execute(); // 仮想メソッドのプライベート・オーバーライド
};
...
tbb::task* RunWorkItem::execute() {
    ReadyPile.runNextWorkItem();
    return NULL;
};
```

RunWorkItem オブジェクトは代替可能です。インテル® TBB のスケジューラーは、これらのオブジェクトを使用して、どの作業項目を行うかではなく、いつ作業項目を行うかを決定します。呼び出しはすべて基本クラス task 経由でディスパッチされるため、仮想メソッド task::execute のオーバーライドはプライベートです。

ほかの優先度手法は ReadyPileType の内部を変更することで実装できます。非常に細粒度の優先度を実装するには、優先度キューを使用することができます。

パターンのスケーラビリティーは ReadyPileType のスケーラビリティーによって制限されるため、この部分には理想的にスケーラブルなコンカレント・コンテナーを使用するべきです。

10 ローカル・シリアライザー

コンテキスト

対話型プログラムを考えます。並列性と応答性を最大限にするために、ユーザーによって要求された操作をタスクとして実装することができます。操作の順序が重要になることがあります。例えば、プログラムがユーザーに編集可能なテキストを提示すると仮定します。ユーザーはテキストを選択する操作や選択したテキストを削除する操作を行います。このとき、同じバッファーで「選択」操作と「削除」操作を逆の順序で行うと問題になります。しかし、それぞれ異なるバッファーで操作を行えば問題はありません。したがって、目標は、異なるオブジェクト間のタスクの順序を制限することなく、オブジェクトと関連するタスクのシリアルな順序を確立することです。

影響

- 特定のオブジェクトと関連する操作はシリアルに実行する必要があります。
- ほかで作業を行うことができるときにスレッドがロックで待機してしまうため、ロックを含む操作のシリアル化は効率的ではありません。

ソリューション

FIFO (先入れ先出し構造) を使用して、作業項目を順番にします。可能であれば、常に項目が処理されるようにします。作業項目が現れたときに処理している項目がない場合、その項目を処理します。処理している項目がある場合は、現れた項目を FIFO にプッシュします。処理している現在の項目が完了したら、FIFO から別の項目をポップして処理します。

このロジックは、FIFO に `concurrent_queue` を使用し、`atomic<int>` で待機および処理している項目数をカウントすることにより、ミューテックスなしで実装できます。この後のサンプルで、処理を詳細に説明します。

サンプル

次のサンプルは、ノンプリエンプティブな優先度のサンプルにローカル・シリアライザーを加えたもので、3 つの優先度とローカル・シリアライザーを実装します。ユーザー・インターフェイスが続きます。

```
enum Priority {
    P_High,
    P_Medium,
    P_Low
};

template<typename Func>
void EnqueueWork( Priority p, Func f, Serializer* s=NULL );
```

表 1 の 3 つの条件が満たされると、テンプレート関数 EnqueueWork はファンクター *f* を実行します。

表 1: 3 つの条件

条件	解決するクラス...
Serializer の事前に行う作業がすべて完了している。	Serializer
スレッドが利用可能。	RunWorkItem
優先度の高い作業は実行する準備ができていない。	ReadyPileType

ファンクターの条件は表の上から下に解決されます。s が NULL の場合、最初の条件は存在しません。EnqueueWork は SerializedWorkItem のファンクターをパッケージして、作業間で最初の条件を解決するクラスに送ります。

```
template<typename Func>
void EnqueueWork( Priority p, Func f, Serializer* s=NULL ) {
    WorkItem* item = new SerializedWorkItem<Func>( p, f, s );
    if( s )
        s->add(item);
    else
        ReadyPile.add(item);
}
```

SerializedWorkItem は WorkItem の派生クラスで、作業の詳細を知ることなく作業の優先度が付けられた部分をパスする方法として提供されます。

```
// 優先度を付ける作業の抽象基本クラス
class WorkItem {
public:
    WorkItem( Priority p ) : priority(p) {}
    // 派生クラスは実際の作業を定義します
    virtual void run() = 0;
    const Priority priority;
};

template<typename Func>
class SerializedWorkItem: public WorkItem {
    Serializer* serializer;
```

```
    Func f;
    /* オーバーライド */
   void run() {
        f();
        Serializer* s = serializer;
        // Serializer の次のファンクターを実行する前に f を破棄します
        delete this;
        if( s )
            s->noteCompletion();
    }
public:
    SerializedWorkItem( Priority p, const Func& f_, Serializer* s ) :
        WorkItem(p), serializer(s), f(f_)
    {}
};
```

基本クラス WorkItem は、ノンプリエンプティブな優先度のサンプルのクラス WorkItem と同じです。シリアル条件の概念は基本クラスから完全に隠されるため、フレームワークはほかの種類の条件や不足している条件を拡張することができます。クラス SerializedWorkItem は本質的にはほかのサンプルの ConcreteWorkItem で、Serializer の観点から拡張したものです。

ファンクターを実行する時間になると、仮想メソッド run() が起動され、次の 3 つのステップを行います。

1. ファンクターを実行します。
2. ファンクターを破棄します。
3. ファンクターが完了し、次の待機ファンクターの条件がなくなったことを Serializer に通知します。

ステップ 3 は、ConcreteWorkItem::run の操作とは異なります。ステップ 3 の後に実行することで並列性を高めることができる場合は、ステップ 2 をステップ 3 の後に実行することもできます。ただし、ステップ 2 にかかる時間がわずかでない場合、次のファンクターを実行する前に完了する必要があるため、示された順に実行することを推奨します。

クラス Serializer はローカル・シリアライザーのコアを実装します。

```
class Serializer {
    tbb::concurrent_queue<WorkItem*> queue;
    tbb::atomic<int> count;          // キューに入れられた項目と実行中の項目のカウント
    void moveOneItemToReadyPile() { // キューから ReadyPile に項目を転送します
        WorkItem* item;
        queue.try_pop(item);
        ReadyPile.add(item);
    }
public:
    void add( WorkItem* item ) {
```

```
        queue.push(item);
        if( ++count==1 )
            moveOneItemToReadyPile();
    }
    void noteCompletion() {          // WorkItem が完了すると呼び出されます
        if( --count!=0 )
            moveOneItemToReadyPile();
    }
};
```

クラスは 2 つのメンバーを保持します。

- 前の作業が完了するのを待つ WorkItem のキュー。
- キューに入れられた作業または実行中の作業のカウント。

操作の順序に注意しながら concurrent_queue<WorkItem*> と atomic<int> を使用することで、ミューテックスの使用を回避しています。カウントの変化はクラス Serializer の動作を理解する鍵です。

- メソッド add が count を 0 から 1 にインクリメントした場合、ほかの作業が実行中でなく、作業を ReadyPile に移動すべきことを示しています。
- メソッド noteCompletion がカウントをデクリメントし、それが 1 から 0 でない場合、キューが空でなく、キューの別の項目を ReadyPile に移動すべきことを示しています。

クラス ReadyPile は、ノンプリエンプティブな優先度のサンプルで説明しています。

優先度が必要ない場合、メソッド moveOneItem の 2 つのバリエーションがあります。バリエーションによって影響は異なります。

- メソッド moveOneItem が item->run() を直接起動します。このアプローチでは、Serializer のオーバーヘッドが相対的に低くなり、スレッドの局所性が高くなります。しかし、これは公平ではありません。タスクの連続したストリームを Serializer が処理する場合、スレッド操作はそのタスクのサービスを続けてほかを除外します。
- メソッド moveOneItem が task::enqueue を起動して、item->run() を起動するタスクをキューに入れます。最初のアプローチよりもオーバーヘッドが高くなり、スレッドの局所性が低くなりますが、飢餓状態 (starvation) を回避できます。

公平性と局所性の高さは相反するものであり、最適な選択肢は状況に依存します。

パターンは、クラス Serializer によって維持されているものよりも一般的な作業項目の条件に一般化します。一般化された Serializer::add は、作業項目に条件があるかどうかを判断して、条件がない場合は直ちに実行します。一般化された Serializer::noteCompletion は、現在の作業項目の完了によって条件がなくなった (以前は条件があった) 項目をすべて実行します。用語「実行」

は、作業を直ちに実行することを意味します。さらに条件がある場合は、次の条件を解決するクラスに作業を転送します。

11 フェンス付きデータ転送

問題

順序に一貫性のあるメモリーモデルがないハードウェアで、メモリーにメッセージを書き込んでいるときに、別のプロセッサーがそのメッセージを読み取る。

コンテキスト

この問題は通常、同期されていない複数のスレッドがメモリーの同じ場所に同時にアクセスしたとき、または読み取りと書き込みを使用して同期を行っているときにのみ発生します。高レベルの同期には通常、好ましくない順序変更を防ぐメカニズムが含まれています。

最近のハードウェアやコンパイラーは、スレッドの操作の順序を保存してメモリー操作の順序を変更することができますが、これはそのスレッドの観点から行われ、ほかのスレッドの観点からではありません。シリアルコードでは、次のコードに示すように、メッセージを書き込んでから準備ができたことをマークします。

```
bool Ready;
std::string Message;

void Send( const std::string& src ) {    // スレッド 1 で実行されます
    Message=src;
    Ready = true;
}

bool Receive( std::string& dst ) {      // スレッド 2 で実行されます
    bool result = Ready;
    if( result ) dst=Message;
    return result;                      // メッセージを受け取ると true を返します
}
```

コードの重要な 2 つの仮定は次のとおりです。

a. Message が書き込まれるまで Ready は true になりません。

b. Ready が true になるまで Message は読み取れません。

これらの仮定は単一プロセッサー・システムではもちろん当てはまります。しかし、マルチプロセッサー・システムでは当てはまりません。ハードウェアやコンパイラーで順序変更を行うと、送

信者の書き込みが順序どおりに行われない (条件 a に当てはまらなくなる) か、受信者の読み取りが順序どおりに行われない (条件 b に当てはまらなくなる) ことがあります。

影響

- 読み取りと書き込みの同期を行います。

関連項目

遅延初期化

ソリューション

メッセージの準備ができたことを知らせるフラグを `bool` から `tbb::atomic<bool>` に変更します。以前のサンプルを修正したサンプル (変更点を青で表示) を次に示します。

```
tbb::atomic<bool> Ready;
std::string Message;

void Send( const std::string& src ) {    // スレッド 1 で実行されます
    Message=src;
    Ready = true;
}

bool Receive( std::string& dst ) {      // スレッド 2 で実行されます
    bool result = Ready;
    if( result ) dst=Message;
    return result;                      // メッセージを受け取ると true を返します
}
```

値 `tbb::atomic` への書き込みには *release* (解放) セマンティクスが用いられ、書き込みを解放する前にその前の書き込みがすべて行われます。値 `tbb::atomic` からの読み取りには *acquire* (取得) セマンティクスが用いられ、読み取りを取得した後にその後の読み取りがすべて行われます。`tbb::atomic` を実装することで、コンパイラーとハードウェアの両方がこれらの順序条件を見ることが保証されます。

バリエーション

高レベルの同期には通常、必要な *acquire* と *release* フェンスが含まれています。例えば、ミューテックスは通常、ロックの取得に *acquire* セマンティクス、ロックの解放に *release* セマンティクスを用いて実装されます。このため、ミューテックスでロックを取得するスレッドは常に、そのミューテックスのロックを解放する前に別のスレッドによって行われるすべてのメモリー書き込みを見ます。

誤ったソリューション

何度も提案される誤ったソリューションについて、なぜそのソリューションが間違っているか理解しておくとよいでしょう。

よくある誤りは、volatile キーワードを使用してフラグを宣言することで問題が解決すると仮定することです。volatile キーワードは直ちに書き込みを行うように制御しますが、一般にほかのメモリー操作に関連してその書き込みの順序には見た目上の効果はありません。この規則の例外は、インテル® Itanium® プロセッサー・ファミリーのプロセッサーです。これらのプロセッサーは、規則により acquire セマンティクスを volatile の読み取り、release セマンティクスを volatile の書き込みに割り当てています。

別の誤りは、条件をテストする前にその条件によって実行されるコードがないと仮定することです。コンパイラーやハードウェアによっては、条件の前にその条件付きコードを投機的に実行する可能性があります。

同様に、プロセッサーがポインターを読む前にポインターのターゲットを読むことができないと仮定することも誤りです。最近のプロセッサーはメインメモリーから個々の値を読むのではなく、キャッシュラインを読みます。ポインターを読む前に、ポインターのターゲットがすでに読み込まれ、キャッシュラインにある場合、プロセッサーがポインターのターゲットを先に読むことは起こりえます。

12 遅延初期化

問題

初期化が初めて必要になったときに初期化を行う。

コンテキスト

データ構造を遅れて初期化することは一般的なテクニックです。使用されていないデータ構造を初期化するコストを回避できるだけでなく、多くの場合、プログラムを構築するための便利な方法となります。

影響

- 複数のスレッドがオブジェクトへのアクセスを共有します。
- オブジェクトは最初にアクセスされるまで作成すべきではありません。

後者には次の理由が含まれます。

- オブジェクトの作成にはコストがかかるため、早く作成するとプログラムのスタートアップが遅くなります。
- 作成したオブジェクトがプログラムのすべての実行で使用されるとは限りません。
- 早期の初期化は、読みやすさや構造上の理由から好ましくないコードを追加することになります。

関連項目

フェンス付きデータ転送

ソリューション

複数の一貫性問題に対処する必要があるため、並列のソリューションは本質的に注意を要します。

競合: 2 つのスレッドが同時に初めてオブジェクトにアクセスし、オブジェクトの作成が行われる場合、両方のスレッドが型 T の同じオブジェクトへの参照を行うように競合を解決する必要があります。

メモリーリーク: 競合が発生した場合、一時的な T オブジェクトがすべてクリーンアップされることを保証しなければなりません。

メモリー一貫性: スレッド X が value=new T() を実行する場合、ほかのすべてのスレッドから見て、割り当て value= の前に new T() によるストアが行われていなければなりません。

デッドロック: もし T() のコンストラクターがロックを取得することを要求し、ロックの現在の保持者も初めてそのオブジェクトにアクセスしようとしている場合はどうすればいいのでしょうか。

2 つのソリューションがあります。ダブルチェック・ロッキング (double-check locking) に基づくソリューションと、コンペアー・アンド・スワップ (compare-and-swap) に依存するソリューションです。トレードオフと問題点の説明は次のサンプルセクションで行います。

サンプル

インテル® TBB での「ダブルチェック」パターンの実装は次のようになります。

```
template<typename T, typename Mutex=tbb::mutex>
class lazy {
   tbb::atomic<T*> value;
   Mutex mut;
public:
   lazy() : value() {}                  // 値を NULL に初期化します
   ~lazy() {delete value;}
   T& get() {
     if( !value ) {                     // 値の読み取りは acquire (取得) セマンティクス
        Mutex::scoped_lock lock(mut);
        if( !value ) value = new T();   // 値の書き込みは release (解放) セマンティクス
     }
     return *value;
   }
};
```

この「ダブルチェック」という名前はパターンが競合を扱う方法に由来します。ロックなしで行われる確認とロックの後に行われる確認があります。最初の確認は、ロックなしで初期化がすでに行われている一般的なケースを扱います。2 つめの確認は、2 つのスレッドがどちらも初期化されていない値を見ていて、どちらもロックを取得しようとしているケースを扱います。この場合、ロックを取得する 2 番目のスレッドは、初期化が行われたことを理解します。

T() が例外をスローした場合、value は NULL のままでオブジェクト lock が破棄されるときにミューテックスがロック解除されるため、ソリューションは正しくなります。

このソリューションはメモリー一貫性問題を正しく扱っています。値 tbb::atomic への書き込みには *release* (解放) セマンティクスが用いられ、書き込みを解放する前にその前の書き込みがすべて行われます。値 tbb::atomic からの読み取りには *acquire* (取得) セマンティクスが用いられ、読み取りを取得した後にその後の読み取りがすべて行われます。これらのプロパティーはどちらもソリューションにとって重要です。書き込みの解放は、T() の構築が値の割り当て前に行われることを保証します。読み取りの取得は、呼び出し元が *value から読み取る場合、"if(!value)" を確

認した後に読み取りが行われることを保証します。解放/取得は本質的にフェンス付きデータ転送パターンです。「メッセージ」は完全に構築されたインスタンス T() で、「準備」フラグはポインター value です。

このソリューションでは、初期化を行っている間スレッドをブロックする必要があります。このため、ブロックに関連する問題が発生することがあります。例えば、最初にロックを取得するスレッドが OS によって休止状態にある場合、ほかのすべてのスレッドはそのスレッドが再開するまで待つ必要があります。ロックフリー・バリエーションでは、競合するすべてのスレッドで初期化を試み、どのスレッドによる試みが成功するかアトミックに決定することで、この問題を回避しています。

インテル® TBB での非ブロック・バリエーションの実装は次のようになります。ロックなしでダブルチェックも行います。

```
template<typename T>
class lazy {
    tbb::atomic<T*> value;
public:
    lazy() : value() {}                    // 値を NULL に初期化します
    ~lazy() {delete value;}
    T& get() {
        if( !value ) {
            T* tmp = new T();
            if( value.compare_and_swap(tmp,NULL)!=NULL )
                // 別のスレッドにより値が設定されたので、自分の値は破棄します
                delete tmp;
        }
        return *value;
    }
};
```

2 つめの確認は式 value.compare_and_swap(tmp,NULL)!=NULL によって行われます。この式は、value==NULL の場合 value=tmp を条件付きで割り当て、以前の value が NULL だった場合 true を返します。このため、複数のスレッドが同時に初期化を試みた場合、compare_and_swap を実行する最初のスレッドは value を T オブジェクトのポインターに設定します。compare_and_swap を実行するほかのスレッドは、非 NULL のポインターが返されることで、一時的な T オブジェクトを削除すべきことがわかります。

ロックによるソリューションのように、メモリー一貫性問題は tbb::atomic のセマンティクスによって扱われます。最初の確認には *acquire* セマンティクスが、compare_and_swap には *acquire* と *release* の両方のセマンティクスが用いられます。

参考文献

読み取りの acquire フェンスを回避する方法は、Mike Burrow のアルゴリズム <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2660.htm> を使用しました。

13 参照カウント

問題

使用されなくなったオブジェクトを破棄する。

コンテキスト

多くの場合、オブジェクトが将来使用されないことがわかっているときは、オブジェクトを破棄することが望まれます。参照カウントは、注意深く行えば並列プログラミングに拡張できる一般的なシリアル・ソリューションです。

影響

- 参照に循環がある場合、その循環が明示的に分解されていなければ、基本的な参照カウントでは不十分です。
- アトミックカウントはハードウェアで相対的にコストがかかります。

ソリューション

スレッドセーフな参照カウントは、インクリメント/デクリメントがアトミックに行われ、デクリメントと「カウントがゼロかどうか」のテストを単一のアトミック操作として行う必要があることを除いて、シリアル参照カウントに似ています。次のサンプルはこのために tbb::atomic<int> を使用しています。

```
template<typename T>
class counted {
    tbb::atomic<int> my_count;
    T value;
public:
    // 単一の参照でオブジェクトを構築します
    counted() {my_count=1;}
    // 参照を追加します
    void add_ref() {++my_count;}
    // 参照を削除し、最後の参照だった場合は true を返します
    bool remove_ref() {return --my_count==0;}
    // オブジェクトへの参照を取得します
    T& get() {
        assert(my_count>0);
        return my_value;
```

```
    }
};
```

カウントがゼロかどうかをテストするために個別の読み取りを使用することは正しくありません。次のコードはメソッド `remove_ref()` の正しくない実装です。2 つのスレッドがデクリメントを実行して `my_count` の読み取りがどちらもゼロになった場合、2 つの呼び出し元はどちらも最後の参照を削除したと誤って伝えられます。

```
    --my_count;
    return my_count==0; // 間違い!
```

オブジェクトが削除される前にすべての保留中の書き込みが完了するように、デクリメントには *release* フェンスを用いる必要があります。

参照カウントが非常に小さな場合、コピーと参照カウントのインクリメントとの間にはタイミング的な問題があるため、アトミックにポインターをコピーしてその参照カウントをインクリメントする単純な方法はありません。このため、別のスレッドがカウントをデクリメントしてオブジェクトを削除することになります。この問題に対応するには、「ハザードポインター」と「pass the buck」の 2 つの方法があります。詳細は、この章の最後にある「参考文献」を参照してください。

バリエーション

アトミック・インクリメント/デクリメントは、通常のインクリメント/デクリメントより桁を大きくすることができますが、よりコストがかかります。冗長なインクリメント/デクリメント操作をなくすシリアル最適化は、アトミックな参照カウントでより重要になります。

ポインターが共有されず対象が共有される場合、重み付けされた参照カウントを使用してコストを減らすことができます。各ポインターに *weight* (重み) を関連付けます。参照カウントは、重みの合計です。ポインター *x* は、x と x′ 間の元の重みを分割することにより、参照カウントを更新しないで、ポインター *x′* としてコピーすることができます。x の重みが小さくなりすぎて分割できない場合、最初に定数 W を参照カウントと x の重みに追加します。

参考文献

D. Bacon および V.T. Rajan、“Concurrent Cycle Collection in Reference Counted Systems”、*Proc. European Conf. on Object-Oriented Programming* (June 2001)。サイクルを収集する参照カウントに基づくガベージコレクターについて説明。

M. Michael、“Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”、IEEE Transactions on Parallel and Distributed Systems (June 2004)。「ハザードポインター」手法について説明。

M. Herlihy、V. Luchangco、および M. Moir、“The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures”、*Proceedings of the 16th International Symposium on Distributed Computing* (Oct. 2002)。「pass the buck」手法について説明。

14 CAS (コンペアー・アンド・スワップ) ループ

問題

プレディケートが満たされるようにスカラー値をアトミックに更新する。

コンテキスト

多くの場合、共有変数は、新しい値が古い値にとって代わるように変更することで、アトミックに更新する必要があります。この変更は、有限状態機械の遷移、または大域的な情報の記録になります。例えば、共有変数は任意のスレッドにおいてこれまでに確認された最大値を記録しているかもしれません。

影響

- 変数は複数のスレッドによって読み取りおよび更新されます。
- ハードウェアは、その種の変数に「コンペアー・アンド・スワップ」を実装します。
- ミューテックスによる更新の保護は回避されます。

関連項目

リダクション

参照カウント

ソリューション

アトミックに現在の値のスナップショットをとり、`atomic<T>::compare_and_swap` を使用して更新します。`compare_and_swap` が成功するまで再試行します。現在の値がある条件と一致した場合は、`compare_and_swap` が成功する前に終了することもあります。

以下のテンプレートは x=F(x) の更新をアトミックに行います。

```
// x=F(x) をアトミックに実行します
template<typename F, typename T>
void AtomicUpdate( atomic<T>& x, F f ) {
    int o;
    do {
        // スナップショットをとります
        int o = x;
        // スナップショットから計算した新しい値のインストールを試みます
    } while( x.compare_and_swap(o,f(o))!=o );
}
```

X の値はほかのスレッドによって変更される可能性があるため、スナップショットをとって中間の計算に使用することが非常に重要になります。

次のコードは、テンプレートを使用して RecordMax によって確認された任意の値のグローバルな最大値を維持する方法を示しています。

```
// UpperBound = max(UpperBound,y) をアトミックに実行します
void RecordMax( int y ) {
    extern atomic<int> UpperBound;
    AtomicUpdate(UpperBound, [&](int value){return std::max(value,y);} );
}
```

y によって UpperBound が増えない場合、AtomicUpdate の呼び出しは冗長な操作 compare_and_swap(o,o) を実行して時間を浪費することになります。一般に、AtomicUpdate のループを F(o)==o の場合に早く終了することにより、この種の冗長な操作を省略できます。特に F==std::max<int> の場合は、テストをさらに単純化できます。RecordMax の次のバージョンでは、単純化されたテストを行っています。

```
// UpperBound =max(UpperBound,y) をアトミックに実行します
void RecordMax( int y ) {  .
    extern atomic<int> UpperBound;
    do {
        // スナップショットをとります
        int o = UpperBound;
        // スナップショットが条件を満たした場合は終了します
        if( o>=y ) break;
        // 新しい値のインストールを試みます
    } while( UpperBound.compare_and_swap(y,o)!=o );
}
```

関与しているすべてのスレッドが共通の場所を修正するため、高い競合により CAS ループのパフォーマンスが低くなることがあります。このため、より効率的なパターンが適用できるかどうかを最初に考慮するべきです。特に次の点を考慮してください。

- 全体的な目的がリダクションの場合は、リダクション・パターンを代わりに使用してください。
- 更新が加算または減算の場合は、atomic<T>::fetch_and_add を使用してください。また、更新が 1 の加算または減算の場合は、atomic<T>::operater++ または atomic<T>::operator-- を使用してください。これらの方法では通常、CAS ループを回避する直接的なハードウェア・サポートが適用されます。

注: compare_and_swap を使用してリンク構造のリンクを更新する場合、「ABA 問題」について理解しておく必要があります。この問題の詳細は、インターネットで検索してください。

全般的な参考文献

このセクションは、全般的な参考文献をリストします。パターン固有の参考文献は、そのパターンの章の最後にリストされています。

- E. Gamma、R. Helm、R. Johnson、J. Vlissides。 *Design Patterns* (1995)。
- Berkeley Pattern Language for Parallel Programming、 http://parlab.eecs.berkeley.edu/wiki/patterns
- T. Mattson、B. Sanders、B. Massingill。 *Patterns for Parallel Programming* (2005)。
- ParaPLoP 2009、 http://www.upcrc.illinois.edu/workshops/paraplop09/program.html
- ParaPLoP 2010、http://www.upcrc.illinois.edu/workshops/paraplop10/program.html
- Eun-Gyu Kim および Marc Snir、"Parallel Programming Patterns"、 http://www.cs.illinois.edu/homes/snir/PPP/index.html