The Parallel Universe

インテル® スレッディング・ビルディング・ブロックの
フローグラフ、
スペキュレーティブ・ロック、
タスク領域

MIT から GCC* メインラインへと進化を遂げた
インテル® Cilk™ Plus

intel® Software

Issue
18
2014

目次

編集者からのメッセージ 3
共通言語によるコーディング
James Reinders

注目記事

MIT から GCC* メインラインへと進化を遂げたインテル® Cilk™ Plus 4
一般的なスレッド・フレームワークよりも高いレベルでの抽象化を提供する、インテル® Cilk™ Plus のタスクおよびデータ並列性を表現する機能を紹介します。

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) のフローグラフ、スペキュレーティブ・ロック、タスク領域 15
フローグラフ・インターフェイス、インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) を利用するスペキュレーティブ・ロック、同時実行制御とワーク分離を強化するユーザー管理タスク領域を含むインテル® TBB の機能を紹介します。

MPI の標準化から 20 年: 共通のアプリケーション・バイナリー・インターフェイスの実現 28
MPI の互換性問題と解決方法、および新しい共通の MPI アプリケーション・バイナリー・インターフェイス (ABI) の可能性を検証します。

新しい MPI-3 標準でパフォーマンスの課題に対処する 33
通信と計算のオーバーラップを測定する方法と、MPI アプリケーションで非ブロッキング集合操作を行う方法を説明します。

OpenMP* 年表 41
OpenMP* のこれまでの歩みを年表で示します。

インテル® アーキテクチャーにおける OpenCL* 投資の活用 42
デバイス間の移植性からインテル® Xeon Phi™ コプロセッサーまで、広範にわたって OpenCL* を利用する方法を紹介します。

© 2014 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Xeon、Xeon Phi、Cilk、VTune は、アメリカ合衆国および / または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。
OpenCL および OpenCL ロゴは、Apple Inc. の商標であり、Khronos の使用許諾を受けて使用しています。

編集者からのメッセージ

James Reinders インテル コーポレーションの並列プログラミング・エバンジェリスト兼ディレクター

Morgan Kaufmann から出版された 2 冊の新しい書籍、『Intel® Xeon Phi™ Coprocessor High Performance Programming』(2013、日本語:『インテル® Xeon Phi™ コプロセッサー ハイパフォーマンス・プログラミング』) および『Structured Parallel Programming』(2012、日本語:『構造化並列プログラミング』) の共著者。このほかに、『Intel® Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism』(O' Reilly Media、2007、日本語:『インテル スレッディング・ビルディング・ブロック ―マルチコア時代の C++ 並列プログラミング』、中国語、韓国語の翻訳版があります) や『VTune™ Performance Analyzer Essentials』 (Intel Press, 2005) などの文献を発表しています。

共通言語によるコーディング

もはや、数学だけが世界の共通語ではありません。「必要は発明の母」という格言のとおり、プログラミング言語とモデルは世界中の開発者とエンジニアによって共有されています。この号では、言語の急速な変化とアプリケーションに革新をもたらす新機能の可能性を紹介します。

注目記事は 2 つあります。1 つ目の「MIT から GCC* メインラインへと進化を遂げたインテル® Cilk™ Plus」では、タスクおよびデータ並列性を表現する C/C++ 言語拡張セットであるインテル® Cilk™ Plus の最新の機能と現在に至るまでの進化を紹介します。2 つ目の「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) のフローグラフ、スペキュレーティブ・ロック、タスク領域」では、主要アルゴリズムを変えたり、再度デバッグしなくても、さまざまな OS やアーキテクチャーへ移植できるクロスプラットフォーム・サポートの拡張と新機能について検証します。

「MPI の標準化から 20 年 : 共通のアプリケーション・バイナリー・インターフェイスの実現」では、2014 年夏に導入される予定の MPI アプリケーション・バイナリー・インターフェイス (ABI) の可能性に触れます。この共通の ABI により、異なる MPI 実装をパフォーマンスに基づいて比較できるようになります。

「新しい MPI-3 標準でパフォーマンスの課題に対処する」では、インテル® MPI ライブラリー 5.0 およびインテル® MPI Benchmarks (IMB) 4.0 でサポートされる、MPI-3 の非ブロッキング集合操作によってもたらされるパフォーマンスの利点について述べます。

「OpenMP* 年表」は、OpenMP* のこれまでの歩みを年表で示します。

最後に、「インテル® アーキテクチャーにおける OpenCL* 投資の活用」は、インテル® Xeon Phi™ コプロセッサーに関するパフォーマンスの可搬性について、考慮すべき点を概説します。

ここで紹介する手法がソフトウェア共通言語を使う上で皆さんのお役に立つことを願っています。ソフトウェア定義データセンター、応答性が要求されるクロスプラットフォームのエクスペリエンス、画期的な新しい API など、ソフトウェアを進化へ導く皆さんの先駆的な取り組みに敬意を表します。

James Reinders
2014 年 6 月

MIT から GCC* メインラインへと進化を遂げたインテル® Cilk™ Plus

Barry Tannenbaum インテル コーポレーション ソフトウェア開発エンジニア

はじめに

インテル® Cilk™ Plus は、タスクとデータの並列性を表現する強力な C/C++ 言語拡張です。実装は簡単で、さまざまなアプリケーションで利用できます。広く使用されているインテル® スレッディング・ビルディング・ブロック (インテル® TBB) と同様、インテル® Cilk™ Plus は MIT の Cilk プロジェクトに端を発しています。インテル® TBB と異なる点は、インテル® Cilk™ Plus はコンパイラーに言語拡張として組込まれており、並列タスクだけでなくベクトル化やリダクション機能を備えていることです。現在、インテル® Cilk™ Plus は、インテル® コンパイラー、GNU* コンパイラー・コレクション (GCC*)、LLVM/Clang のブランチでサポートされています。

長年にわたりソフトウェア開発者はムーアの法則に頼って、より高速かつ多くの処理を実行できるアプリケーションを作成してきました。1965 年、インテルの共同設立者である Gordon Moore は、集積回路のトランジスター数は 2 年ごとに倍増すると述べました。このムーアの法則により、初期のインテル® 4004 から今日のデータセンターを支えるインテル® Xeon® プロセッサーへと、コンピューターはより高速に、そして高性能になりました。しかし、2000 年代半ば、従来の汎用プロセッサーはある壁に突き当りました。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

クロックレートの増加により、利点よりも問題 (消費電力など) のほうが顕著になったためです。その結果、インテルや他のプロセッサー・メーカーは、ムーアの法則で予測されるトランジスター数の増加を利用して、複数のコアやベクトル命令ユニットの導入に踏み切りました。以来、プロセッサー・コアの数とベクトルユニットのベクトル幅は着実に増加しています。今日、インテルは最大 61 コアを搭載し、各コアに 16 の単精度浮動小数点操作または 8 の倍精度浮動小数点操作を同時処理するベクトルユニットを備えた、インテル® Xeon Phi™ コプロセッサー製品ファミリーを製造しています。

現代のプロセッサーで利用可能なリソースを活用する、ベクトル化されたマルチスレッド・アプリケーションのプログラミングは複雑で面倒なものです。経験豊富な開発者でさえも、同時に起こり得るすべてのことを把握することは難しいでしょう。その結果、再現と修正が困難な問題を抱え、コア数の増加に伴ってスケーリングしないプログラムになりがちです。インテル® Cilk™ Plus は、開発者が最近のプロセッサー・アーキテクチャーを最大限に活用する強力なアプリケーションの開発、ビルド、デバッグを簡単に行えるように設計されています。

インテル® Cilk™ Plus の簡単な例

インテル® Cilk™ Plus の代表例として、フィボナッチ数列を計算する次のアプリケーションがあります。これは、フィボナッチ数列を求める最良の実装とはいえませんが、再帰関数の良い例です。

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

この例は、インテル® Cilk™ Plus によるタスク並列処理を示しています。fib() の 1 つ目の再帰呼び出しは、継続される処理 (2 番目の fib(n-2)) と並列に実行できます。どちらの再帰呼び出しも関数の最後 (`cilk_sync` の位置) で結果が結合され、リターンされる前に完了しなければなりません。インテル® Cilk™ Plus は簡単なループ構造でも利用できますが、特にこのような再帰アルゴリズムに適しています。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

行列乗算は、インテル® Cilk™ Plus によるデータ並列処理の一例です。以下の例は単純なシリアル実装です。

```
template<typename T>
void matmul(int ii, int jj, int kk, T *a, T *b, T *product)
{
    for (int i = 0; i < ii; ++i)
        for (int j = 0; j < jj; ++j)
            for (int k = 0; k < kk; ++k)
                product[i * jj + j] += a[i * kk + k] * b[k * jj + j];
}
```

以下は、同じコードの内側のループを配列表記でベクトル化したものです。

```
template<typename T>
void matmul_vec(int ii, int jj, int kk, T a[], T b[], T product[])
{
    for (int i = 0; i < ii; ++i)
        for (int j = 0; j < jj; ++j)
            product[i * jj + j] =
                __sec_reduce_add(a[i*kk:kk:1] * b[j:kk:jj]);
}
```

a[i*kk:kk:1] * b[j:kk:jj] は、行列 a の行の各要素を行列 b の列に掛けて、要素数 **kk** の一時ベクトルを取得します。そして、ビルトイン関数 **__sec_reduce_add()** の呼び出しで一時ベクトルの要素を合計します。

インテル® Cilk™ Plus の起源

インテル® Cilk™ Plus は、元々 MIT コンピューター科学研究所の 3 つの研究プロジェクトでした。1994 年にこれら 3 つのプロジェクトが統合され、C 言語でスレッドを利用する "Cilk" プロジェクトが誕生しました。C 言語拡張である MIT Cilk はソースツーソース・コンパイラーとして実装され、1994 年 9 月に Cilk-1 システムとして初めてリリースされました。現在の実装は Cilk-5.3 で、MIT コンピューター科学・人工知能研究所 (CSAIL) から提供されていますが、サポートはすでに終了しています。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

2006 年、商用の C++ 言語拡張の実装を開発するため、Cilk Arts, Inc. が MIT から Cilk テクノロジーのライセンスを受け、2008 年 12 月に Visual Studio* (Windows*) と GCC*/C++ (Linux*) をサポートする Cilk++ v1.0 をリリースしました。残念なことに、この Cilk++ は非標準のリンク形式を採用していたため C/C++ コードからの呼び出しが複雑で、標準デバッガーを使用するのも困難でした。

2009 年 7 月 31 日、Cilk Arts, Inc. はインテルに買収され、その製品とエンジニアリング・チームはインテルの傘下に入りました。インテルと Cilk Arts, Inc. の協力により、Cilk テクノロジーはさらなる進化を遂げ、2010 年 10 月、インテル® C++ Composer 12.0 の一部としてインテル® Cilk™ Plus が登場しました。インテル® Cilk™ Plus では、これまでの Cilk 実装のすべての機能を C と C++ で利用できます。また、標準の呼び出し規約を採用しており、既存のデバッガーと互換性があり、データ並列処理をサポートする構文が追加されています。

インテルは、インテル® Cilk™ Plus をさらに改良して業界全体に広く普及させたい意向を表明しています。その取り組みの一環として、2010 年 11 月にインテル® Cilk™ Plus の言語拡張仕様、ABI 仕様、アノテーション仕様を公開し、ほかのベンダーがインテル® Cilk™ Plus とインテル® Cilk™ Plus ランタイム・ライブラリーを実装できるようにしました。

2011 年には、GCC* C/C++ コンパイラーの "cilkplus" ブランチでインテル® Cilk™ Plus の実装を支援し、2012 年に完成した最初の実装は GNU Tools Cauldron 2012 で発表されました。このブランチで、インテルはインテル® Cilk™ Plus ランタイムをオープンソースとしてリリースし、以降、GCC* コミュニティーと協力してインテル® Cilk™ Plus が GCC* のメインラインに組込まれるように働きかけてきました。その結果、GCC* 4.9 向けのトランクにインテル® Cilk™ Plus が組み込まれることになりました。

開発者にとってインテル® Cilk™ Plus が GCC* C/C++ コンパイラーでサポートされることはどのような意味があるでしょう？

- オープンソースのインテル® Cilk™ Plus を利用することができます。インテル® Composer XE 固有の機能に依存することなく、インテル® Cilk™ Plus をコードで使用し、インテル® コンパイラーに関係なく、インテル® Cilk™ Plus を GCC* で継続して利用できます。
- インテル® Cilk™ Plus ランタイムの完全なソースを含むインテル® Cilk™ Plus 実装を入手したり、製品に対する意見や要望を送ったり、製品の向上に協力することができます。インテルではインテル® Cilk™ Plus のコミュニティー Web サイトを開設し、インテル® Cilk™ Plus の拡張と向上、ほかのアーキテクチャーやオペレーティング・システムへの移植を推進するべく、オープンソース・コミュニティーに協力をお願いしています。

インテル® Cilk™ Plus を使用する理由

インテル® Cilk™ Plus は、インテル® TBB や OpenMP* などのスレッド・フレームワークよりも高いレベルの抽象化を提供します。開発者は、アプリケーションを効率良く並列に実行する作業よりも、アプリケーションの潜在的な並列性を表現することに注力できます。

インテル® Cilk™ Plus ランタイムは、ワークスチール・スケジューラーによってすべてのプロセッサー ・コアで

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

アプリケーションを効率良く実行します。何らかの理由でコアがストールしたり、ワークロードのバランスが取れていない場合、スケジューラーはそのコアからタスクをスチールし、アイドル状態のコアで実行します。並列性の表現と並列実行のスケジュールを分離することで、十分な並列性が表現されている場合 (十分な数のタスクがある場合)、アプリケーションはコードを変更しなくてもコア数に応じてスケーリングします。

インテル® Cilk™ Plus を使用することで、開発者は構成可能なコードを記述できます。「構成可能」とは、インテル® Cilk™ Plus でライブラリーを記述し、システムリソースのオーバーサブスクリプションを引き起こすことなくそのライブラリーをインテル® Cilk™ Plus アプリケーションから呼び出せることを意味します。

インテル® Cilk™ Plus のレデューサーにより、ロックを使用することなく競合状態を解決することができます。

インテル® Cilk™ Plus ランタイムとインテル® Cilk™ Plus レデューサーは連携してシリアル・セマンティクスを提供します。適切に記述された決定論性のあるインテル® Cilk™ Plus アプリケーションは、実行するコアの数に関係なく同じ結果を出力します。シリアル・セマンティクスによってシリアル実行の結果と並列実行の結果を直接比較できるため、アプリケーションのテストが容易になります。さらに、シングルコアで実行し、標準デバッガーでプログラムの状態を確認できるのでアプリケーションのデバッグも簡単です。

インテル® Cilk™ Plus のベクトル化拡張は、開発者にデータ並列性を表現する手段を提供します。最近のプロセッサーでベクトルユニットを最大限に利用しないことは、計算能力のほとんどをアイドル状態のままにしておくことを意味します。

「インテル® Cilk™ Plus のワークスチール・スケジューラーは、できるだけ多くのコアへタスクを自動的にスケジュールし、プロセッサー間の負荷バランスをほぼ最適にします。」

インテル® Cilk™ Plus の機能

- タスクキーワード : アプリケーションのタスク並列処理を簡潔に表現します。
- レデューサー : 必要に応じて自動的に共有変数の " ビュー " を作成し、ロックを使わずに " レデュース " 操作を行うことで、タスク間の共有変数の競合を排除します。
- #pragma simd: ベクトル化されるべきループを示します。
- 配列表記 : 配列と部分配列のデータ並列処理を表現できます。
- SIMD 対応関数 : 配列表記または #pragma simd ループから暗黙に呼び出される関数のベクトル化バージョンを生成します。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

タスクキーワード

インテル® Cilk™ Plus には、**cilk_spawn**、**cilk_sync**、**cilk_for** の 3 つのキーワードがあります。これらのキーワードで、アプリケーションの並列領域を記述します。インテル® Cilk™ Plus のワークスチール・スケジューラーは、できるだけ多くのコアへタスクを自動的にスケジュールし、プロセッサー間の負荷バランスをほぼ最適にします。

cilk_for

cilk_for ループは、各反復を並列に実行できることを除いて、標準の for ループと同じです。ループに続く処理を実行する前に、すべての反復が完了することが保証されます。インテル® Cilk™ Plus ランタイムは、すべてのコアを最適に使用するように自動的にループを分割します。[1]

cilk_spawn および cilk_sync

cilk_spawn は、関数の呼び出し元と、呼び出し先の関数を並列に実行できることをランタイムシステムに通知します。

cilk_sync は、現在の関数からスポーンされたすべての関数呼び出しが完了するまで、この位置で待機する必要があることを通知します。

cilk_spawn と **cilk_sync** を利用して再帰アルゴリズムを簡単に実装することができます。この 2 つのキーワードをプログラムに追加すると、インテル® Cilk™ Plus ランタイムはその領域を実行するプロセッサー・コアを選択します。

インテル® Cilk™ Plus キーワードは、プログラムの並列性を表すだけであって、その領域が必ずしも並列化されることを意味しません。インテル® Cilk™ Plus ランタイムは、利用可能なプロセッサー・リソースに応じて、**cilk_for**、**cilk_spawn**、および **cilk_sync** で表現された並列領域をスケジュールします。適切に記述されたインテル® Cilk™ Plus プログラムはシングルコアでも正しく実行されます。

一部のスレッド・フレームワークと異なり、インテル® Cilk™ Plus では、プログラムを実行するコア数に合わせてコードをカスタマイズするべきではありません。特定のコア数で最適に実行するようにアプリケーションをチューニングすることは危険です。コア数やキャッシュメモリー容量が変わったり、システムで別のプログラムが同時に実行される場合、再度チューニングが必要になります。一般に、インテル® Cilk™ Plus ランタイムがビジー状態のコアからタスクをスチールし、アイドル状態のコアで実行できるようにするには、プログラムにコア数の少なくとも 10 倍の並列タスクがなければなりません。

インテル® Cilk™ Plus キーワードは、完全に厳密なフォーク・ジョイン型並列処理を実装します。つまり、**cilk_for** ループや関数内のスポーンはすべて、**cilk_for** または関数が終了する前に完了します。そのため、考慮しなければならないコードを限定でき、プログラムの並列性を判断しやすくなります。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

レデューサー

プログラムを並列化することで、シリアルプログラムで生じるすべての問題に加えて競合状態が発生する可能性が出てきます。競合状態とは、2 つの並列コードが同じメモリー位置にアクセスし、少なくとも一方が書き込みを行う状態です。その結果、不定の動作が生じます。競合状態を解決する典型的な方法はロックで共有変数を保護することですが、ロックには次のような問題があります

- ロックの誤用はデッドロックを引き起こすことがあります。
- ロックで保護された領域で競合が発生すると、並列プログラムが遅くなります。
- ロックは実行順序を制御できないため、適切に使用されても、決定論性のない結果になることがあります。

レデューサーは並列領域ごとに個別の変数のビューを作成し、ロックを使わずにそれぞれのビューを更新できるため、これらの問題をすべて解決します。並列領域が終了する際に、それぞれビューはプログラムをシリアル実行した場合と同じ順序でマージされます。

#pragma simd

#pragma simd は、後に続く for ループをベクトル化するようにコンパイラーに指示します。何らかの理由によりコンパイラーがそのループをベクトル化できない場合、警告メッセージを出力するかどうかを指定できます。#pragma simd は、ループを安全にベクトル化できることをコンパイラーに知らせます。一方、#pragma ivdep などのベクトル化の " ヒント " は、安全なことを証明できる場合のみベクトル化するようコンパイラーに指示し、ベクトル化の可能性を制限します。これは、安全であっても単一ソースのみを参照してコンパイルする場合、コンパイラーが安全性を証明できないとベクトル化が適用されないことを意味します。

配列表記

インテル® Cilk™ Plus には、配列全体または部分配列に対する高レベルの操作を表現する表記があります。これらの表記は、コンパイラーが効率良くベクトル化できるように支援します。配列表記は、固定長配列と可変長配列のどちらにも使用できます。以下に配列表記の構文を示します。

ポインター [下限 : 要素数 : ストライド]

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

コロンで区切られた 3 つのセクション・パラメーターには、任意の整数式を指定できます。セクションの要素が配列オブジェクトの境界内にあることを保証するのは開発者の責任です。デフォルト値を使用する場合、各セクション・パラメーターは省略できます。

- デフォルトの < 下限 > は 0 です。デフォルトの < 下限 > を使用する場合、< 要素数 > と < ストライド > もデフォルト値にしなければなりません。
- デフォルトの < 要素数 > は配列の要素数です。コンパイラーが配列の要素数を判断できない場合、要素数を指定する必要があります。デフォルトの < 要素数 > を使用する場合、< ストライド > もデフォルト値にしなければなりません。
- デフォルトの < ストライド > は 1 です。デフォルトの < ストライド > を使用する場合、2 つ目の ":" は省略します。

ここで、< 要素数 > は部分配列で処理される要素の数です。**(< 要素数 > * < ストライド >) + < 下限 >** が配列の境界内になるように保証するのは開発者の責任です。

配列表記

```
int a[20], b[20], c[20];

// 配列を初期化します。
a[:]= 5;


// 配列要素に 5 を足します。
b[:]= a[:]+ 5;


// 偶数要素を設定します。
b[0:10:2] = 15;


// 2 つの配列を合計します。
c[:]= a[:]+ b[:];


// 配列の各要素に対して
// 関数を呼び出します。
func(a[:]);
```

等価な C/C++ シリアルコード

```
int a[20], b[20], c[20];

// 配列を初期化します。
for (int i = 0; i < 20; i++)
    a[i] = 5;

// 配列要素に 5 を足します。
for (int i = 0; i < 20; i++)
    b[i] = a[i] + 5;

// 偶数要素を設定します。
for (int i = 0; i < 20; i += 2)
    b[i] = 15;

// 2 つの配列を合計します。
for (int i = 0; i < 20; i++)
    c[i] = a[i] + b[i];

// 配列の各要素に対して
// 関数を呼び出します。
for (int i = 0; i < 20; i++)
    func(a[i]);
```

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

以下に、配列表記によって簡潔に表現できるコード例を示します。部分配列は if 文、C/C++ の条件演算子、または case 文で利用できます。

```
if (a[0:n] > b[0:n])
    c[0:n] = a[0:n] - b[0:n];
else
    c[0:n] = 0;
```

この例は、**a[i] > b[i]** かどうかに応じて、**c[i]** を **a[i] – b[i]** または 0 に設定します。

インテル® Cilk™ Plus には、部分配列に対して操作を行い、スカラー値を返すリダクション関数があります。ビルトインのリダクション関数は、要素の合計、積、最大値、最小値、最大 / 最小要素のインデックス、すべての要素がゼロ / 非ゼロかどうか、ゼロ / 非ゼロの要素があるかどうかを返します。

SIMD 対応関数

SIMD 対応関数は、ベクトル化されたループで処理されるスカラー引数、部分配列、配列要素のいずれかに対して呼び出すことができる関数です。Windows* では **__declspec(vector)**、Linux* または Mac* OS では **__attribute__((vector))** アノテーションで宣言され、コンパイラーに関数のスカラーバージョンと SIMD バージョンの両方を生成するように指示します。部分配列で呼び出される場合、SIMD 対応関数はベクトル幅分の要素を受け取り、それらを同時に処理します。次に例を示します。

```
__declspec(vector) func(double a);

func(a[:]);
```

まとめ

カラツバ法の多項式乗算は、分割統治アプローチにより、2 つの多項式が十分に小さくなるまで再帰的に多項式のペアを 2 等分して乗算します。http://www.cilkplus.org/download にあるサンプルコードでは、カラツバ法の 4 つの実装 (シリアル実装、配列表記によりベクトル化された実装、インテル® Cilk™ Plus キーワードにより並列化された実装、配列表記によりベクトル化されインテル® Cilk™ Plus キーワードにより並列化された実装) が提供されています。配列表記とインテル® Cilk™ Plus を併用する実装では、タスクとデータの並列処理による相乗効果が得られます。ベクトル化されたバージョンでは、わずかなスピードアップしかもたらされません。これは、インテル® コンパイラーの自動ベクトル化により、このアプリケーションがすでにベクトル化されているためで

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

す。

```
Testing Karatsuba implementations...Validated

Starting speed tests.Parallel runs will use 8 threads...

Timing 2048 multiplications of 10000-degree polynomials

                Version   Time   Speedup

                 Serial 18.267   1.00 x

             Vectorized 16.334   1.12 x

               Parallel  2.792   6.54 x

   Parallel/Vectorized  2.590   7.05 x
```

コンパイラー (インテル® 64)	コンパイラー・オプション	システム構成
インテル® C++ Composer XE 2013 SP1 Update 2 Windows* 版	リリース構成 /O3 /Qipo /fp:fast /QxHost	Windows* 7 2 x インテル® Xeon® プロセッサー E5520 (2.27GHz) 6GB メモリー ハイパースレッディング無効 (BIOS 設定)

ここで紹介したカラツバ法の実装のサンプルコードは、インテル® Cilk™ Plus Web サイト のダウンロード・ページ (http://www.cilkplus.org/download) (英語) から入手できます。

GCC* 以外のサポート

GCC* の実装に加えて、インテルは LLVM/Clang でインテル® Cilk™ Plus を実装するべく取り組んでいます。この実装は GitHub* で公開されており、http://cilkplus.github.io/ (英語) から入手できます。この記事の執筆時点では、インテル® Cilk™ Plus/LLVM はまだ開発中です。

さらに、インテルは、オリジナルの Cilk を開発した MIT グループと協力して、インテル® Cilk™ Plus のソフトウェア・パイプラインの実装に取り組んでおり、そのプロトタイプがインテル® Cilk™ Plus Web サイトの Experimental Software ページ (英語) で公開されています。

実際に使ってみる

インテル® C++ Composer XE ユーザーは、プログラムにインテル® Cilk™ Plus キーワードを追加するだけです。必要に応じて、コンパイラーにより自動的にインテル® Cilk™ Plus ランタイムにリンクされます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

GCC* 4.9 のダウンロードおよびビルド方法は、インテル® Cilk™ Plus Web サイト (http://www.cilkplus.org/build-gcc-cilkplus) (英語) を参照してください。GCC* ユーザーは、次のオプションでインテル® Cilk™ Plus 機能を有効にし、インテル® Cilk™ Plus ランタイムにリンクします。

```
g++ -fcilkplus -lcilkrts
```

LLVM/Clang のインテル® Cilk™ Plus ブランチのダウンロードおよびビルド方法は、http://cilkplus.github.io/ (英語) を参照してください。

関連ドキュメント (英語)

インテル® Cilk™ Plus コミュニティー Web サイト (http://www.cilkplus.org/; http://www.isus.jp/article/intel-cilk-plus/): インテル® Cilk™ Plus に関する情報、サンプル・アプリケーション、パブリック・ライブラリー、最新のソースなど。

インテル® Cilk™ Plus チュートリアル (http://www.cilkplus.org/cilk-plus-tutorial): インテル® Cilk™ Plus の概要。

配列表記 C/C++ 拡張のプログラミング・モデル (https://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/optaps/common/optaps_par_cean_prog.htm): 配列表記、SIMD 対応関数、#pragma simd の使用法に関する詳細。

インテル® Cilk™ Plus の LLVM/Clang 実装 (http://cilkplus.github.io/): インテル® Cilk™ Plus/LLVM 開発ページ。実装のステータス、インテル® Cilk™ Plus/LLVM のビルド方法に関する情報とリンク。

インテル® C++ コンパイラーのコードサンプル (http://software.intel.com/en-us/code-samples/intel-c-compiler): 明示的なベクトル化と並列化を使用する C++ サンプル。

1. この記事の執筆時点で、インテル® Cilk™ Plus の cilk_for 機能はまだ GCC* 4.9 に実装されていません。インテルは GCC* コミュニティーと協力してこの機能の実装を完了させるため取り組んでいます。

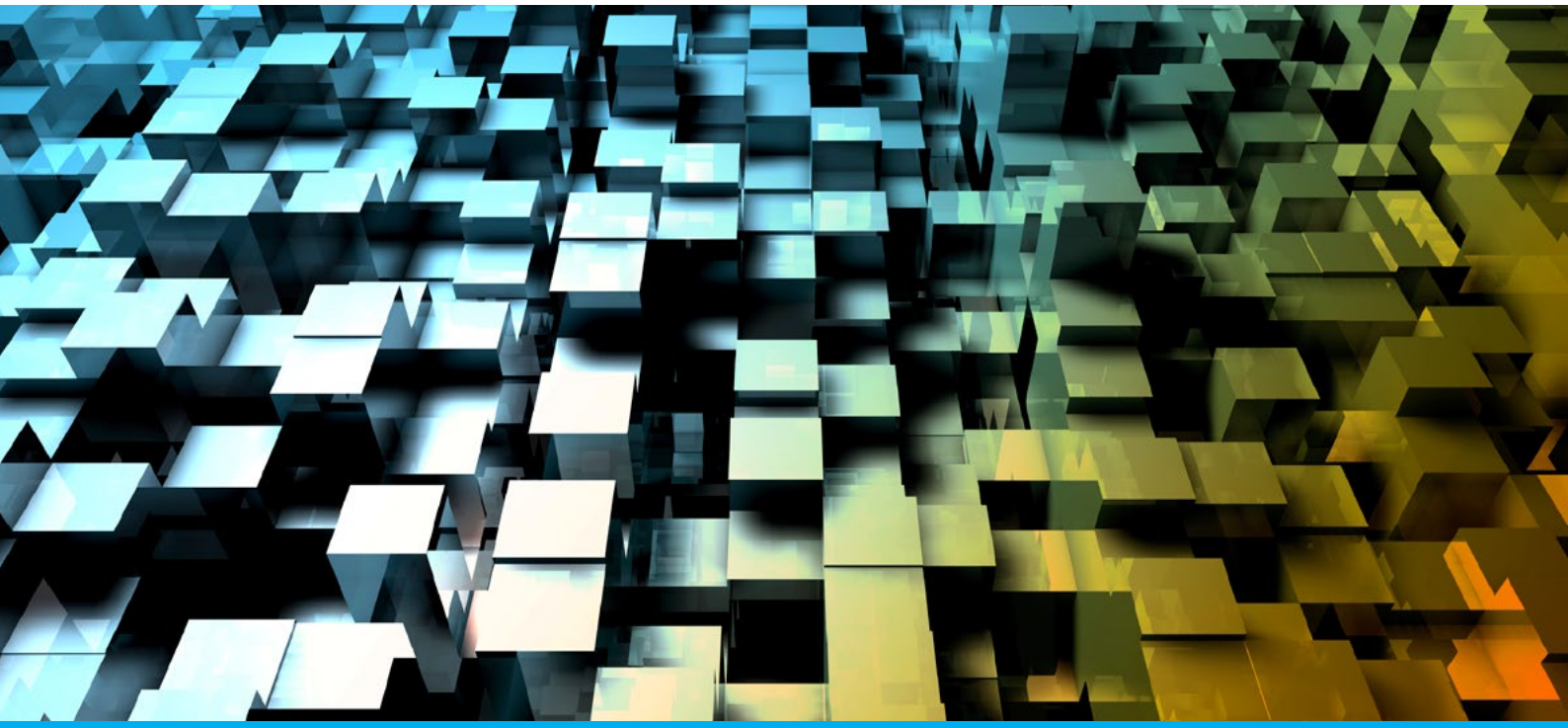
インテル® Cilk™ Plus に関する日本語情報サイト : http://www.isus.jp/article/intel-cilk-plus/

インテル® Cilk™ Plus を試そう

次のソフトウェアで利用可能:
インテル® C++ コンパイラー
インテル® Parallel Studio XE
インテル® Composer XE

インテル® Cilk™ Plus を評価する >
インテル® Parallel Studio XE を評価する >

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



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) のフローグラフ、スペキュレーティブ・ロック、タスク領域

Alexey Kukanov インテル コーポレーション ソフトウェア & サービスグループ 開発製品部門
スレッディング・ランタイム担当ソフトウェア・アーキテクト
Vladimir Polin 同ソフトウェア ・エンジニアリング・マネージャー
Michael J. Voss 同ソフトウェア・アーキテクト

はじめに

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) ライブラリーは、ソフトウェア開発者に C++ アプリケーションおよびライブラリーを並列化するソリューションを提供します。インテル® TBB ライブラリーのよく知られた利点は、ループやタスクを使うアプリケーションの並列パフォーマンスとスケーラビリティーを簡単に向上できることです。インテル® TBB には多くの汎用並列アルゴリズム、フローグラフ・インターフェイス、ワークスチール・ベースのタスク・スケジューラー、コンカレント・コンテナー、スケーラブルなメモリー割り当て、スレッド・ローカル・ストレージ、同期プリミティブが含まれています **(図 1)**。基本的な並列パターンをこれらのビルディング・ブロックにマップすることで、プラットフォームの詳細とスレッド化のメカニズムを抽象化し、コア数の増加に応じてパフォーマンスがスケーリングする、優れた並列アプリケーションを作成できます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

1 インテル® スレッディング・ビルディング・ブロックの機能

よく使用される並列パターンは、『Structured Parallel Programming [1]』(日本語 :『構造化並列プログラミング』) で説明されています。これらのパターンに対応するインテル® TBB アルゴリズムを以下に示します。

`parallel_for`: マップ

`parallel_reduce`、**`parallel_scan`**: レデュース、 スキャン

`parallel_do`: ワークパイル

`parallel_pipeline`: パイプライン

`parallel_invoke`、**`task _ group`**: フォーク・ジョイン

`flow_graph`: プラミング (リアクティブ / ストリーミング・アプリケーション)

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® TBB ライブラリーは、あらゆる C++ 互換コンパイラーで利用できます。特定のファンクター・クラスの実装が不要なため、コーディングを軽減するラムダ式などの C++11 機能もサポートしています。各インテル® TBB ライブラリー・コンポーネントは、個別に使用したり、あるいは他のスレッド化テクノロジーと組み合わせて利用できます。

インテル® TBB 4.2 では、インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) や Windows* 向けインテル® Xeon Phi™ コプロセッサーなどの最新のインテル® アーキテクチャー機能のサポートに加え、Windows* ストア・アプリケーションおよび Android* アプリケーションのサポート [2,3]、いくつかの新機能と改善されたポピュラーな機能を提供します (詳細は、インテル® TBB ドキュメントを参照)。インテル® TBB 4.2 はスタンドアロン製品として、あるいはインテル® INDE、インテル® Cluster Studio XE、インテル® Parallel Studio XE、インテル® C++ Studio XE、インテル® Composer XE、インテル® C++ Composer XE などのソフトウェア・バンドルのコンポーネントとして利用できます。

インテル® TBB ライブラリーのオープンソース版はコミュニティーの協力を経て、さらに多くのアーキテクチャー、コンパイラー、オペレーティング・システムをサポートしています。インテル® アーキテクチャーおよび SPARC* 互換プロセッサー向け Solaris* + Oracle* C++ コンパイラー、IBM* Blue Gene* スーパーコンピューター、PowerPC* 互換アーキテクチャー、Android*、Windows Phone* 8、ARM* 互換アーキテクチャー向け Windows RT*、FreeBSD*、ROS (Robot Operating System)、そして GCC* の組込みアトミック命令をサポートするその他多数のプラットフォームで動作します。**このクロスプラットフォーム・サポートにより、インテル® TBB ライブラリーを使用するアプリケーションは、主要アルゴリズムを変更したり、再度デバッグしなくてもさまざまな OS に移植することができます。**

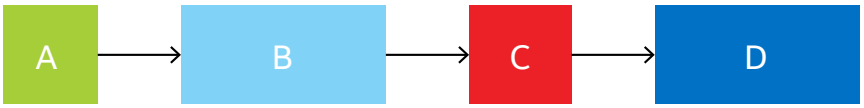
ここでは、インテル® TBB のいくつかの機能に注目します。最初に、インテル® TBB 4.0 以降に含まれる**フローグラフ・インターフェイス**の概要を説明します。次に、インテル® TBB 4.2 で追加された 2 つの新機能を紹介します。1 つ目は**スペキュレーティブ・ロック**で、インテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) を利用します。2 つ目は**ユーザー管理タスク領域**で、同時実行制御とワーク分離を強化します。

依存関係グラフとデータフロー・グラフ

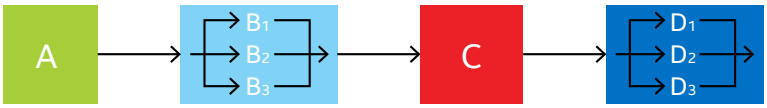
インテル® TBB ライブラリーはループベースの並列アルゴリズムがよく知られていますが、フローグラフ・インターフェイス [4] は依存関係グラフとデータフロー・アルゴリズムを効率良く実装し、開発者がより高いレベルでアプリケーションを並列化できるようにします。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

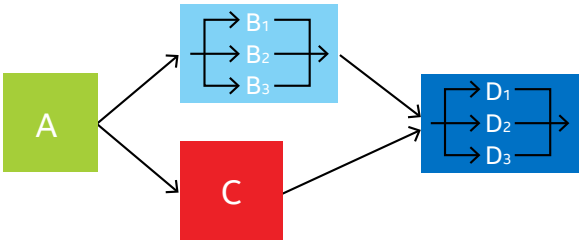
例えば、**図 2(a)** に示す 4 つの関数をシーケンシャルに呼び出す簡単なアプリケーションについて考えてみます。ループ並列アプローチは各関数を調査し、**parallel_for** や **parallel_reduce** などのアルゴリズムを適用できる並列性を探します。これだけで優れたパフォーマンスが得られる場合もありますが、このレベルでは十分な並列処理がないこともあります。例えば、**図 2(b)** から、関数 B と D でループ並列処理が見つかり、それにより実行時間が向上することが分かります。しかし、さらにパフォーマンスを向上するにはどうすれば良いでしょうか？



(a) シーケンシャル実行



(b) ループ並列実行



(c) グラフおよびループ並列実行

2 並列化の異なる形式を示す簡単な例

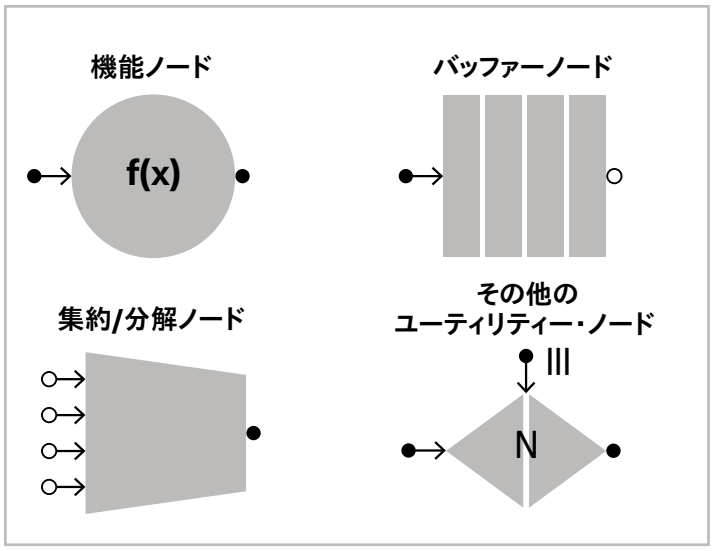
場合によっては、関数を 1 つずつ順番に呼び出すことは大きな制約になります。半順序が必要な場合であっても、呼び出しの全順序を指定する必要があるためです。**図 2(a)** では、開発者によって、必要な値がすべて計算されてから各関数が実行されるように順序付けられています。しかし、関数 B と C が A によって生成される出力を必要とし、C は B の出力に依存しない場合はどうでしょうか？その場合のグラフおよびループ並列の実装を **図 2(c)** に示します。この実装では、ループレベルの並列性が表され、全順序の代わりに B と C を同時に実行可能な半順序が指定されています。

インテル® TBB のフローグラフ・インターフェイスにより、開発者はグラフ並列性を簡単に表現できます。メディア、ゲーム、金融、ハイパフォーマンス・コンピューティング、医療などのさまざまな分野における依存関係、ストリーミング、データフロー・グラフをサポートする機能を提供するこのインターフェイスは、インテル® TBB 4.0 以降で利用できます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

フローグラフでは、計算はノードで、計算の通信チャネルはエッジで表されます。インテル® TBB スケジューラーがグラフで表された並列性を利用できるように、開発者はノードをスケジュールする際に考慮すべきすべての依存関係をエッジで表す必要があります。ノードがメッセージを受け取ると、そのメッセージのボディー・オブジェクトを実行するインテル® TBB タスクがスポーンされます。

フローグラフ・インターフェイスは、ユーザー定義のボディー・オブジェクトを実行する機能ノード、メッセージの順序付けとバッファーへの格納を行うバッファーノード、メッセージの結合 / 分割を行う集約 / 分解ノード、その他の特殊ノードなど、さまざまな種類のノードをサポートします (**図 3**)。開発者はこれらのノードをエッジで繋ぎ合わせて、依存関係を指定したり、ワークを実行するボディー・オブジェクトを提供します。



3 フローグラフ・インターフェイスでサポートされるノードの種類

以下は、簡単な "Hello World" フローグラフ・アプリケーションのソースコードです。このコードは並列性を含みませんが、インターフェイス構文の使用例を示します。"Hello" と "World" を出力するラムダ式により、hello と world という 2 つのノードが作成されます。各ノードは **`continue_node`** という、インターフェイスで提供される機能ノードです。**`make_edge`** 呼び出しは **hello** ノードと **world** ノード間のエッジを作成します。**hello** ノードによりスポーンされた関数が完了すると、**hello** ノードは **world** ノードにメッセージを送り、**world** ノードはラムダ式を実行するタスクをスポーンします。

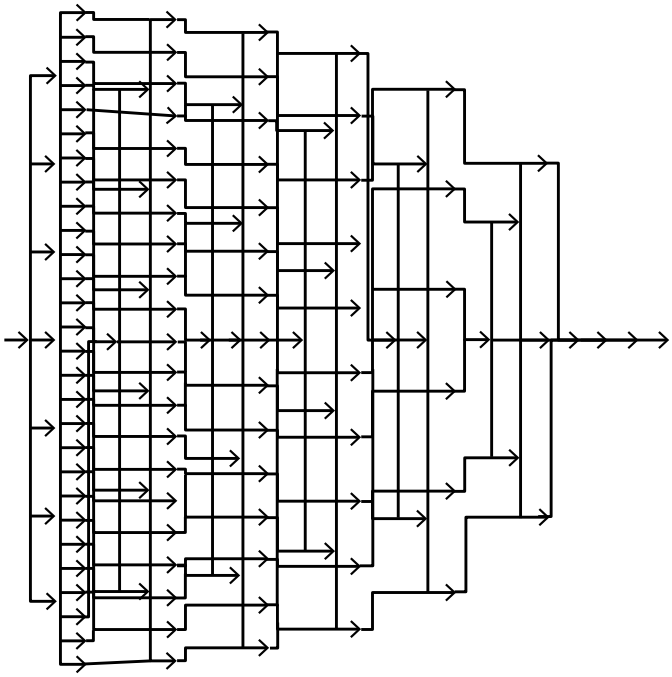
インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

```
#include “tbb/flow_graph.h”
#include <iostream>

using namespace std;
using namespace tbb::flow;

int main() {
    graph g;
    continue_node< continue_msg> hello( g,
        [ ]( const continue_msg &) {
            cout << “Hello”;
        }
    );
    continue_node< continue_msg> world( g,
        [ ]( const continue_msg &) {
            cout << “ World\n”;
        }
    );
    make_edge(hello, world);
    hello.try_put(continue_msg());
    g.wait_for_all();
    return 0;
}
```

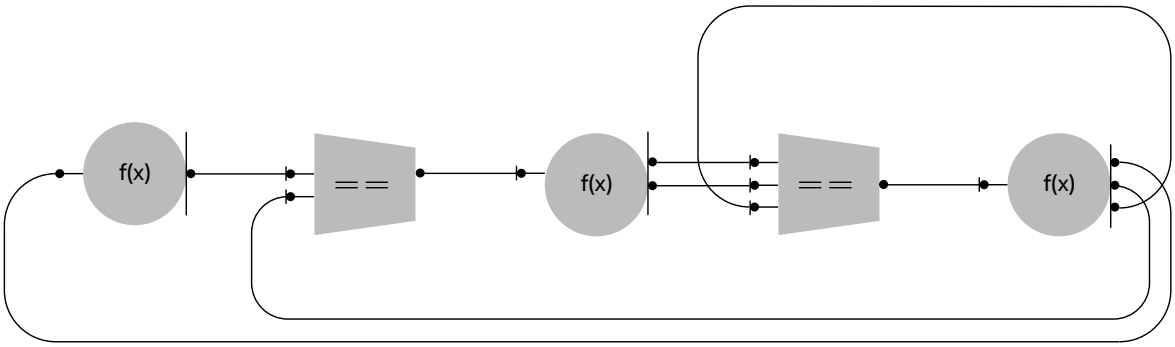
上記のコードで **hello.try_put(continue_msg())** 呼び出しは **hello** ノードにメッセージを送り、**hello** ノードはボディー・オブジェクトを実行するタスクをスポーンします。そのタスクが完了すると、**hello** ノードは **world** ノードにメッセージを送ります。ノードによってスポーンされたすべてのタスクが完了すると、**g.wait_for_all()** 呼び出しはリターンします。

インテル® TBB のフローグラフ・インターフェイスは、数千ものノードとエッジ、循環、バッファーなどを含む非常に複雑なグラフを表せます。**図 4** はコレスキー分解の 2 つのフローグラフの実装 (『Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems』[5] で説明されているアルゴリズムに似たもの) を表現できます。**図 4(a)** のフローグラフで、インテル® マス・カーネル・ライブラリー (インテル® MKL) 関数 dpotf2m、dtrsm、dgemm、dsyrk の呼び出しは個別のノードとして表されています。この例では行列のタイルごとのノード数が多く、グラフが大きくなっていますが、オリジナルのシーケンシャルなコレスキー分解のループの入れ子構造で、関数呼び出しをノードとエッジの作成に置き換えるだけでグラフを作成できます。このグラフでは、関係の前後の通信にエッジが使用されており、各ノードはすべてのプレデセッサーが完了するまで待機する必要があります。このグラフから並列性を簡単に見つけることができます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



(a) 依存関係グラフを使用する場合



(b) データフロー・グラフを使用する場合

4 フローグラフ・インターフェイスを使用するコレスキー分解の 2 つの並列実装

図 4(b) は、フローグラフ・インターフェイスで表せる別の実装です。1 つのノードですべてのインテル® MKL 関数呼び出しに対応するこのバージョンのほうがコンパクトです。この実装ではノード間のメッセージとしてタイルが渡されます。一致するタイルのセットを受け取ると、ノードはそのセットにボディー・オブジェクトを適用するタスクをスポーンし、新しいタイルを生成してそれを別のノードに送ります。このグラフの並列性は、各フローグラフ・ノードの複数のインスタンスを同時に実行できるインテル® TBB の機能によってもたらされます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

図 4 の実装の詳細は本題から外れるためここでは取り上げませんが、この例から、インテル® TBB のフローグラフ・インターフェイスがいかに強力で柔軟性のある抽象化であるかが分かるでしょう。**図 4(a)** のようにインテル® MKL の呼び出しごとに個別のノードを作成する DAG (Directed Acyclic Graph) を作成することも、**図 4(b)** のように循環、結合、条件付き実行を含むコンパクトなデータフロー・グラフを作成することもできます。

フローグラフ・インターフェイスの詳細は、『インテル® TBB リファレンス・マニュアル』を参照してください。

スペキュレーティブ・ロック

インテル® TBB 4.2 の機能であるスペキュレーティブ・ロックは、インテル® TSX ベースの新しい同期クラスです。スペキュレーティブ・ロックは、データの読み取り / 書き込みが競合しないという仮定の下に、保護されたクリティカル・セクションを同時に実行することができます。競合が発生した場合、1 つ以上のスペキュレーティブ実行が無効になり、保護されたデータは変更されず、ほかのスレッドからは利用できないままになります。競合が発生したスレッドはクリティカル・セクションを再度実行し、場合によっては (インテル® TSX はスペキュレーティブ実行が最終的に成功することを保証していないため) 通常のロックを取得してデータを保護します。

スペキュレーティブ・ロックのインテル® TBB 実装 [6, 7] ではこれらの処理はすべて内部で行われ、開発者は通常の mutex API を使用できます。また、インテル® TSX に対応していないプロセッサーでは自動的に通常のロックが使われるため、対応プロセッサーではインテル® TSX を利用し、そうでない場合は利用しない可搬性の高いプログラムを作成することができます。

インテル® TBB ライブラリーは現在、インテル® TSX をサポートする **`speculative_spin_mutex`** および **`speculative_spin_rw_mutex`** の 2 つの mutex クラスを提供しています。**`speculative_spin_rw_mutex`** はインテル® TBB 4.2 Update 2 でプレビュー機能として追加されました。

`speculative_spin_mutex` クラスは API 互換で、**`spin_mutex`** クラスによく似ています。どちらも同じヘッダーファイル (**`tbb/spin_mutex.h`**) にあります。大きな違いはインテル® TSX に対応しているかどうかとデータサイズです。ほかのデータとキャッシュラインを共有することで起こる実行の競合 (フォルス・シェアリング) とパフォーマンス低下を避けるため、**`speculative_spin_mutex`** は 2 つのキャッシュラインを占有します。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

以下は、インテル® TBB のスペキュレーティブ・ロックを使用するコード例です。

```
#include <tbb/spin_mutex.h>
#include <set>
tbb::speculative_spin_mutex tsx_mtx;
std::set<int> g_Set;
void thread_safe_add_to_set( int value ) {
    tbb::speculative_spin_mutex::scoped_lock lock(tsx_mtx);
    g_Set.insert(value);
}
```

speculative_spin_rw_mutex クラスは、名前から想像できるようにスペキュレーティブ実行で読み取り / 書き込みスピンロックを実装します。どのスペキュレーティブ・ロックも本質的に排他的ではなく、読み取りだけでなく、競合しない書き込みも同時に行えます。そのため、「スペキュレーティブ RW ロック」は類語反復のように思えるかもしれません。しかし、スレッドは「実際に」ロックを取得する可能性があります。その場合、**speculative_spin_mutex** はデータの読み取り / 書き込みに関係なく、スレッドに排他的なアクセスを付与する必要があるため、実際には読み取り / 書き込みロックではありません。一方、**speculative_spin_rw_mutex** は常に複数の読み取りを許可します。さらに、実際の読み取りとスペキュレーティブな読み取りを同時に実行できます。しかし、これにはクラスの内部データフィールドを個別のキャッシュラインに保持しなければならないというコストが伴います。フォルス・シェアリングを回避するためにも、**speculative_spin_rw_mutex** は 3 つのキャッシュラインを占有します。

speculative_spin_rw_mutex は現在 **tbb/spin_rw_mutex.h** ヘッダーファイルにあり、実装で **spin_rw_mutex** を使用していますが、この 2 つのクラスは完全に互換性があるわけではありません。**speculative_spin_rw_mutex** には **lock()** メソッドと **unlock()** メソッドがありません。代わりに、スコープ・ロック・パターンを強制します。つまり、**speculative_spin_rw_mutex::scoped_lock** クラスを介してアクセスしなければなりません。これはまだプレビュー機能なので、ヘッダーファイルをインクルードする前に TBB_PREVIEW_SPECULATIVE_SPIN_RW_MUTEX マクロを非ゼロ値に設定する必要があります。

残念ながら、スペキュレーティブ・ロックの適用方法と利点は問題ごとに異なります。この新しいクラスは「スピンロックよりも優れている」と考えるべきではありません。パフォーマンスを慎重に測定し、特定のケースにスペキュレーティブ・ロックが適しているかどうかを判断してください。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

ユーザー管理タスク領域

インテル® TBB ライブラリーに追加されたもう1つの新機能は、ユーザー管理タスク領域です。

タスク領域とは、スレッドがタスクを共有し、スチールする場所を指します。当初、インテル® TBB ライブラリーはアプリケーションで 1 つのグローバル領域のみをサポートしていました。その後、異なるスレッドによるワークは分離したほうが良いというフィードバックに基づいて、アプリケーション・スレッドごとに個別のタスク領域を保持するように変更されました。さらに、同時実行制御とワーク分離をアプリケーション・スレッドに関連付けないで欲しいという要望があり、これに対応するためユーザー管理タスク領域が追加されました。現在、これはまだプレビュー機能ですが (利用するには TBB_PREVIEW_TASK_ARENA マクロを非ゼロ値に設定する必要があります)、2014 年後半には完全にサポートされる予定です。

ユーザー管理領域の API は **task_arena** クラスによって提供されます。**task_arena** の作成時に、ユーザーは同時実行可能なスレッド数とアプリケーション・スレッド用に予約する数を指定できます。

```
#define TBB_PREVIEW_TASK_ARENA 1
#include <tbb/task_arena.h>
tbb::task_arena my_arena(4, 1);
```

上記の例では、4 つのスレッドでタスク領域が作成されます (そのうち 1 つは常にアプリケーション・スレッド用に予約されています)。つまり、インテル® TBB ライブラリーで管理される最大 3 つのワーカースレッドが、このタスク領域で共有されるタスクを処理できます。**task_arena** にジョブを追加できるアプリケーション・スレッドの数に制限はありませんが、ここでは同時実行可能なスレッドは 4 つに制限されています。「ほかの」すべてのスレッドは、このタスク領域にあるタスクを実行することはできません。

タスク領域にワークを追加するには、**execute()** メソッドまたは **enqueue()** メソッドを呼び出します。

```
my_arena.enqueue( a_job_functor );
my_arena.execute( a_job_functor2 );
```

これらのメソッドに渡すジョブは C++11 ラムダ式、またはファンクター・クラスのインスタンスで表せます。この 2 つのメソッドは、タスク領域へのジョブの追加方法が異なります。**task_arena::enqueue()** は「開始した後は忘れてよい (fire-and-forget)」ジョブを追加する非同期呼び出しであり、呼び出し元のスレッドはタスク領域に参加せずに直ちにリターンします。一方、**task_arena::execute()** は追加したジョブが完了するまでリターンしません。可能であれば、呼び出し元のスレッドはタスク領域にあるタスクを実行します。そうでない場合、呼び出し元のスレッドはジョブが完了するまでブロックされます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

task_arena に多くのシリアルジョブを追加することは可能ですが、そのような使用方法は想定されていません。通常、**parallel_for** 呼び出しなどを使用する、十分な並列性を備えたジョブを追加します。

```
my_arena.execute( [&]{
    tbb::parallel_for(0,N,iteration_functor());
});
```

あるいは、フローグラフを使用するものを追加します。

```
tbb::flow::graph g;
...// ここでグラフを作成
my_arena.enqueue( [&]{
    ...// グラフの計算を開始
});
...// その他の処理
my_arena.execute( [&]{
    g.wait_for_all();
}); // フローグラフが完了するまでリターンしない
```

タスク領域の詳細は、『インテル® TBB リファレンス・マニュアル』を参照してください。

まとめ

豊富なコンポーネントを備えるインテル® TBB の C++ テンプレート・ライブラリーは、より高いレベルでタスクベースの並列性を効率良く利用し、将来のマルチコアとメニーコアの能力を活かす移植性に優れたスケーラブルな実装を支援します。このライブラリーを利用することで、開発者は並列性に関する下位レベルの詳細ではなく、アプリケーションで並列性を表現することに専念できます。インテル® TBB ライブラリーは、最もよく使われる高レベルの並列アルゴリズムとコンカレント・コンテナーの効率良くハイパフォーマンスな実装に加えて、スケーラブル・メモリー・アロケーター、ロック、アトミック操作などの低レベルのビルディング・ブロックも提供します。

インテル® TBB ライブラリーは長い実績を誇る製品ですが、インテルは継続的にインテル® TBB のパフォーマンス向上と機能拡張に取り組んでいます。インテル® TBB 4.0 では、開発者が依存関係グラフとデータフローグラフをより簡単に表現できるようにフローグラフが追加されました。インテル® TBB 4.2 では、新しいアーキテクチャー機能であるインテル® TSX ベースの新しい同期クラスと、同時実行とワーク分離の制御に関するユーザーからの要望に対応するためユーザー管理タスク領域が追加されました。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

最新のインテル® TBB バージョンと製品情報は、 公式サイトとオープンソース・サイトをご覧ください。

インテル® TBB 公式サイト (英語): **http://software.intel.com/en-us/intel-tbb**

インテル® TBB オープンソース・サイト (英語): **http://threadingbuildingblocks.org**

インテル® TBB ドキュメント (英語): **http://software.intel.com/en-us/tbb_4.2_ug**

インテル® TBB フォーラム (英語): **http://software.intel.com/en-us/forums/intel-threading-building-blocks**

インテル® TBB 日本語情報サイト : **http://www.isus.jp/article/intel-tbb**

参考文献

1. Structured Parallel Programming (日本語 :『構造化並列プログラミング』), Michael McCool, Arch Robison, and James Reinders, 2012.
   http://parallelbook.com/ (英語)
2. Android* チュートリアル : 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) を使用するマルチスレッド・アプリケーションの記述 , Vladimir Polin, 2013.
   http://www.isus.jp/article/idz/android/android-tutorial-writing-a-multithreaded/
3. Windows* 8 Tutorial:Writing a Multithreaded Application for the Windows Store* using Intel® Threading Building Blocks, Vladimir Polin, 2013.
   http://software.intel.com/en-us/blogs/2013/01/14/windows-8-tutorial-writing-a-multithreaded-application-for-the-windows-store-using (英語)
4. The Intel® Threading Building Blocks Flow Graph, Michael J. Voss, Dr. Dobb’ s, October 2011.
   http://www.drdobbs.com/tools/the-intel-threading-building-blocks-flow/231900177 (英語)
5. Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems, Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc, 2010 Symposium on Parallel & Distributed Processing (IPDPS), April 2010.
6. Transactional Memory Support: The speculative_spin_mutex, Christopher Huson, 2013.
   http://software.intel.com/en-us/blogs/2013/10/07/transactional-memory-support-the-speculative-spin-mutex (英語)
7. Transactional Memory Support: The speculative_spin_rw_mutex, Christopher Huson, 2014.
   http://software.intel.com/en-us/blogs/2014/03/07/transactional-memory-support-the-speculative-spin-rw-mutex-community-preview (英語)

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

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) を評価する >

BLOG HIGHLIGHTS

インテル® Xeon Phi™ コプロセッサー開発者向けリソースガイド

TAYLOR KIDD »

インテル® メニー・インテグレーテッド・コア (インテル® MIC) アーキテクチャー・ベースのインテル® Xeon Phi™ コプロセッサーで初めて作業するソフトウェア開発者を支援するため、役割別に次の 3 つのガイドが提供されています。

- 管理者向けガイド : インテル® Xeon Phi™ コプロセッサーが装着されたサーバー (およびそのサーバーのクラスターを含む) を管理する方を対象にしています。
- 開発者向けガイド: インテル® Xeon Phi™ コプロセッサーが装着されたシステム向けにプログラミングする方を対象にしています。
- 一般向けガイド : インテル® Xeon Phi™ コプロセッサーについて詳しく知りたい方、特にインテル® Xeon Phi™ コプロセッサーを導入するかどうかを決定するために情報が必要な方を対象にしています。

各ガイドには、それぞれの対象者に役立つ情報が記載されています。例えば、クラスターの保守に関する情報は管理者には役立ちますが、開発者にはあまり必要ないでしょう。同様に、プログラミング構文やセマンティクスは開発者にとっては重要ですが、通常、管理者は知る必要がありません。

この記事の続きはこちらで ご覧になれます。 >

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

MPI の標準化から 20 年 : 共通のアプリケーション・バイナリー・インターフェイスの実現

Alexander Supalov インテル コーポレーション クラスター・ツール・アーキテクト
Artem Yalozo インテル コーポレーション ソフトウェア・エンジニア

はじめに

分散メモリー・コンピューティングの業界標準であるメッセージ・パッシング・インターフェース (MPI)[1] が誕生してから今年で 20 年になります。ハイパフォーマンス・コンピューティング (HPC) で要求される高度なパフォーマンスにおいて、MPI に勝るインターフェイスはほとんどないでしょう。

1994 年 5 月の MPI Forum で発表されて以来、MPI は大きな進化を遂げました。2012 年 9 月には最新のメジャーバージョンである MPI-3 標準 [2] がリリースされました。このバージョンでは、高速な一方向通信、非ブロッキング集合操作、その他の機能が追加されています。

この最新バージョンに基づいて、インテルは SuperComputing 2013 にて Argonne National Laboratory、Cray Corporation、および IBM Corporation と協力し、長い間待ち望まれていた共通の MPI アプリケーション・バイナリー・インターフェイス (ABI) を作成するためのイニシアチブを結成しました。この ABI は、インテル® MPI ライブラリー 5.0 として 2014 年 6 月にリリースされ、MPICH、Cray* MPI、IBM* POE を含む多くの互換サードパーティー製品でも利用できます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

目的

ハイパフォーマンス・コンピューティング・クラスターで利用されるマルチファブリック・メッセージ・パッシング・ライブラリーであるインテル® MPI ライブラリーにとって、互換性は長年にわたる大きな課題でした。[3] ユーザーは、製品のメジャーバージョン間でさえも常に互換性が保持されることを期待しています。インテル® MPI ライブラリーでは新しいバージョンへ簡単に移行できるようにバイナリー互換のアップグレード・パスを用意しています。ABI 互換でなければ、ライブラリーの新しいバージョンがリリースされるたびに、古いバージョンで開発されたリリース済みソフトウェアをすべて再リリースしなければなりません。

潜在的な互換性問題の原因

以前のインテル® MPI ライブラリーの互換性問題は、異なるソースベースを使用していたことが原因でした。各バージョンで異なる MPICH[4] ソースコードを使用し、異なるバージョンの MPI 標準を実装していました。例えば、インテル® MPI ライブラリー 4.1 は MPICH2、つまり MPI-2.2 標準をベースにしていました。インテル® MPI ライブラリー 5.0 は MPICH3、つまり新しい MPI-3 標準をベースにしています。

アプリケーション・バイナリー・インターフェイスは、2 つのプログラムモジュール間の低レベルのインターフェイスです。ライブラリーへのバイナリー・インターフェイスを定義するものであり、アプリケーション・プログラミング・インターフェイス (API) とは異なります。ABI は関数の呼び出し方法、データ型のサイズ、レイアウト、アライメントを定義します。同じ ABI を利用することで、互換プログラムは同じランタイム規則に従います。プログラムがライブラリーの以前のバージョンにリンクされている場合、ライブラリーはバイナリー互換なので、再コンパイルしなくても新しいバージョンで動作します。

また、新しいバージョンのライブラリー関数はすべて、以前のバージョンと同じ動作になります。当然のことながら、以前のバージョンで動作が正しくない場合は、新しいバージョンでも動作が変わることがあります。これにより動作の互換性が失われる可能性があります。さらに、MPI 標準の変更によって互換性が失われることもあります。

互換性のシナリオ、問題、解決方法

ソフトウェアの下位互換性と上位互換性の両方が保証されることが理想的ですが、実際には下位互換性と新しい標準の実装要件の間で合理的なバランスがとられます。

インテル® MPI ライブラリーで重要なのは下位互換性です。新しいバージョンでビルドされたアプリケーションは、以前のバージョンのランタイムで動作できなければなりません。上位互換性も考慮されますが、保証されません。

ABI の互換性問題はクライアント・コードでクラッシュを引き起こしたり、データを破損する可能性があります。また、問題に気付かないまま見過ごされることもあるでしょう。そのため、テスト時に互換性問題によるクラッシュを見つけるのではなく、その前に潜在的な互換性問題を注意深く調査すべきです。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® MPI ライブラリーには C、C++、および Fortran 77/90 に固有の機能がいくつかあります。各インターフェイスへの変更は、バイナリー互換性問題を引き起こす可能性があります。表 1 に、インテル® MPI ライブラリー 5.0 (MPICH3 ベース) と 4.1 (MPICH2 ベース) の間で見つかっているバイナリー互換性問題の原因と解決方法を示します。

原因 (インテル® MPI ライブラリー 4.1 と MPICH3 間の変更)	言語固有	解決方法
事前定義定数の変更 : **MPI_MAX_ERROR_STRING**	すべて	インテル® MPI ライブラリー 4.1 の ABI に戻します。
定数/新しい定義の追加: **MPICH_ATTR_FAILED_PROCESSES** **MPI_UNWEIGHTED** **MPICH_ERR_LAST_CLASS**	C	定数を含め、新しい定義を使用します。
事前定義定数の追加/変更: **MPI_COMBINER_HINDEXED_BLOCK** **MPI_COMBINER_***	C Fortran	**HINDEXED_BLOCK** に次の値を割り当て、残りはそのままにします。
MPI_Status 項目の順序変更 (**count** と **cancelled** が下に移動)。**count** の型の **int** から 64 ビットの **MPI_Count** への変更。 **typedef struct MPI_Status** { int MPI_SOURCE; int MPI_TAG; int MPI_ERROR; **MPI_Count count;** **int cancelled;** } MPI_Status;	C	**count** と **cancelled** を上に移動します。 32 ビットの **count** と cancelled を 63 ビットの **count** と 1 ビットの **cancelled** 値として解釈します。
事前定義定数の追加/削除: **MPI_2COMPLEX**、 **MPI_2DOUBLE_COMPLEX**	Fortran	定数を含めます。
共通ブロックの削除/追加: **MPIPRIV1**、**MPIPRIV2**、**MPIPRIVC**、**MPIFCMB5**、**MPIFCMB9**	Fortran	インテル® MPI ライブラリー 4.1 の ABI に戻し、MPI-3 用の新しい共通ブロックをすべて追加します。
仮想関数テーブルの変更 (順序変更): **virtual void Shift()** **Cartcomm Dup()** **virtual int Get_cart_rank()** **virtual void Get_topo()** **virtual int Get_dim()** **virtual int Map()** **virtual Cartcomm Sub()**	C++	インテル® MPI ライブラリー 4.1 の ABI に戻します。

1 バイナリーの互換性問題と解決方法

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® MPI ライブラリー 5.0 では、既知の互換性問題が修正され、インテル® MPI ライブラリー 4.x とバイナリー互換性を実現しました。当初、ベータ版インテル® MPI ライブラリー 5.0 は MPICH3 と互換性がありませんでしたが、ビルド段階で ABI 変更を調整して MPICH3 互換ライブラリーをビルドできることが分かったため、MPICH3 との互換性が保持されました。

表 2 に、インテル® MPI ライブラリー 4.1 (MPICH2 1.4 ベース) と 4.0 (MPICH2 1.1 ベース) の間で見つかっている動作の互換性問題の原因と解決方法を示します。

原因 (インテル® MPI ライブラリー 4.1 と MPICH2 1.4 間の変更)	**言語固有**	**解決方法**
MPI 標準における一部の関数の動作変更: **MPI_Cart_create** **MPI_Cart_map** **MPI_Graph_create** **MPI_Win_get_attr**	C	I_MPI_COMPATIBILITY 環境変数の値に基づいて関数の動作を変更します。
MPI 標準における一部の集合操作の動作変更: **Gather** **Allgather** **など**	C	I_MPI_COMPATIBILITY 環境変数の値に基づいて関数の動作を変更します。

2 動作の互換性問題と解決方法

インテル® MPI ライブラリーは新しい標準を実装しなければならず、新しい標準では動作が変更される可能性があるため、デフォルトでは動作の互換性を保証していません。ただし、実行時に **`I_MPI_COMPATIBILITY` 環境変数**を 4 に設定することで、動作の下位互換性を確保することができます。

これにより、1 つのライブラリーで新しい標準をサポートしつつ、以前のバージョンでビルドされたほとんどのアプリケーションと下位互換性を保持できます。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

まとめ

インテル® MPI ライブラリーは、すべての過去のバージョンと互換性があります。これは、インテル® MPI ライブラリーの優れた利点と言えるでしょう。新しい MPI ABI は、2014 年の夏にパートナー各社で導入される予定です。新しい ABI の詳細は、MPICH ABI Compatibility Initiative[5] の Web サイト (http://www.mpich.org/abi/) を参照してください。導入後、インテル® MPI ライブラリー、MPICH、Cray* MPI、および IBM* POE は実行時に完全互換になります。さらに、この ABI により、1 つの MPI ライブラリーで MPI-2.x と MPI-3 の両方をサポートできるようになります。これは、以前の MPI 標準と関連アプリケーションに対する投資を保護し、1 つのパッケージをサポートするだけで済むため、MPI 実装とその保守が容易になります。また、パートナー各社の MPI 実装にわたってバイナリー互換性が拡張されることも、顧客とパートナーの双方にとって重要です。

ABI によりライブラリー間の切り替えが可能になるため、開発者は異なる MPI 実装をパフォーマンスに基づいて比較することができます。さらに、現在特定の MPI ライブラリー向けにビルドされているツールがほかの MPI ライブラリーでも利用できるようになるでしょう。

参考文献

1. メッセージ・パッシング・インターフェイス (MPI) ドキュメント : http://www.mpi-forum.org/docs (英語)
2. メッセージ・パッシング・インターフェイス・フォーラム。
   MPI: A Message-Passing Interface Standard, Version 3.0.
   September 21, 2012.
   http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (英語)
3. インテル® MPI ライブラリー : www.intel.com/go/mpi (英語) ;
   http://www.isus.jp/article/idz/hpc/intel-mpi-library/
4. MPICH: http://www.mpich.org (英語)
5. MPICH ABI Compatibility Initiative: http://www.mpich.org/abi/ (英語)

インテル® MPI ライブラリーを試そう >

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

新しい MPI-3 標準でパフォーマンスの課題に対処する

Mikhail Brinskiy インテル コーポレーション ソフトウェア開発エンジニア
Alexander Supalov インテル コーポレーション クラスター・ツール・アーキテクト
Michael Chuvelev インテル コーポレーション ソフトウェア・マネージャー
Evgeny Leksikov インテル コーポレーション

はじめに

インテル® MPI ライブラリー 5.0 およびインテル® MPI Benchmarks (IMB) 4.0 でサポートされる MPI-3 の非ブロッキング集合操作はパフォーマンスの利点をもたらします。ここでは、通信と計算のオーバーラップを測定する方法と、MPI アプリケーションで非ブロッキング集合操作を行う方法を紹介します。

メッセージ・パッシング・インターフェイス (MPI) 標準は、分散メモリーシステムでよく使用されるプログラミング・インターフェイスです。最新の MPI-3 標準には、非ブロッキング / 隣接集合操作、リモート・メモリー・アクセス (RMA) インターフェイスの拡張、ラージカウントのサポート、新しいツール・インターフェイスなどの主要な新機能があります。ラージカウントのサポートは大量のデータをシームレスに処理し、高速な一方向操作はリモート・メモリー・アクセス (RMA) ベースのアプリケーションを高速化し、非ブロッキング集合操作はアプリケーションの計算処理と通信処理をオーバーラップさせることでパフォーマンスを向上します。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

ここでは、非ブロッキング集合操作 (NBC) と新しい RMA インターフェイスの 2 つの MPI-3 機能に注目します。NBC による通信と計算のオーバーラップの効果を検証し、新しい RMA 機能の利点を説明します。

パフォーマンスの測定には、MPI-3 標準をサポートするベータ版インテル® MPI ライブラリー 5.0[3] とベータ版インテル® MPI Benchmarks 4.0 (IMB) を使用しました。ベータ版 IMB 4.0 には非ブロッキング集合操作の効果を測定する `IMB-NBC` と、MPI-3 における RMA インターフェイスの重要なアップデートである `IMB-RMA` の 2 つの新しいバイナリーが追加されています。IMB と各ベンチマークの説明は、**http://software.intel.com/en-us/articles/intel-mpi-benchmarks (英語)** を参照してください。

非ブロッキング集合操作 (NBC)

非ブロッキングのポイントツーポイント操作のように、NBC セマンティクスにより通信と計算をオーバーラップさせることができます。また、NBC は、データ依存によって多くの集合アルゴリズムで生じる疑似同期の影響を緩和します。非ブロッキング集合操作の可能性については、**http://htor.inf.ethz.ch/publications/img/hoefler-nbc-standard.pdf (英語)** に詳しい分析があります。

一般に非ブロッキング操作 (集合およびポイントツーポイント) では、ユーザーの介入なしにバックグラウンドで処理が行われます。[1] しかし、MPI 標準では明示的に処理規則が定義されていないため、「高品質な」実装により非同期処理が行われます。そのため、以前の MPI ライブラリーの多くは非同期処理を提供せず、単に 1 つの MPI 呼び出しと別の MPI 呼び出しの間ですべての通信を実行していました。NBC の登場により、定期的に `MPI_Test()` ルーチンを呼び出して明示的にメッセージを処理するか、あるいは `MPI_Wait()` を 1 回呼び出して処理を完了することで、集合操作と通信をオーバーラップさせるという選択肢が増えました。

IMB は `MPI_Test()` または `MPI_Wait()` 呼び出しを通じて、よく使われる非ブロッキング通信、特に操作の開始と完了の間にアクティビティーが終了するケースにおいて、通信と集合操作のオーバーラップの効果を測定します。この手法では、ネットワークやオペレーティング・システム (OS) のノイズによって生じる不正確さを最小限に抑えられるため、通信時間と計算時間はほぼ同じであると仮定しています。IMB は、完全にベクトル化された 100x100 行列のベクトル積を計算することで、任意の時間 CPU をビジー状態にします。以下に、ベンチマークの処理の流れを示します。

1. 通信呼び出しにかかる時間 (`MPI_Ibcast()` から `MPI_Wait()` まで) を測定します。
2. 通信を開始します (`MPI_Ibcast()` を呼び出します)。
3. ステップ 1 で測定した時間分の計算を開始します。これにより、通信と計算にかかる時間がほぼ同じになるようにします。
4. 通信が完了するのを待ちます (`MPI_Wait()` 呼び出しを待機します)。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

IMB-NBC ベンチマークは、上記の処理を実行して 4 つの測定時間を出力します。

> **time_pure** – CPU アクティビティーと同時に実行されなかった非ブロッキング操作の時間

> **time_CPU** – テスト CPU アクティビティーの時間 (time_pure に近い値になります)

> **time_ovrlp** – CPU アクティビティーと同時に実行された非ブロッキング操作の時間

> **overlap** – 次の式で予測されたオーバーラップの割合

```
overlap = 100 * max(0, min(1, (time_pure + time_CPU - time_ovrlp) /
max(time_pure, time_CPU)))
```

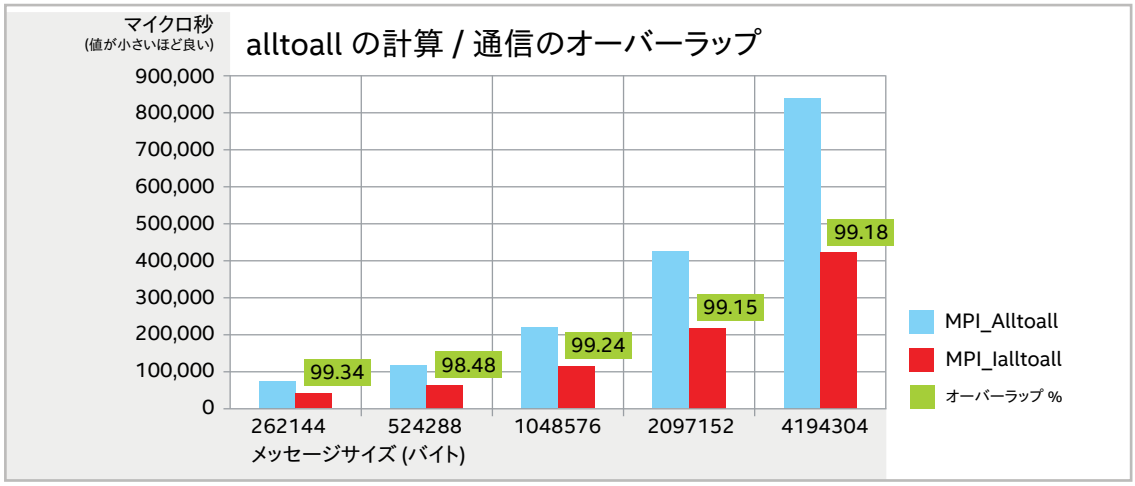
インテル® MPI ライブラリーは、マルチスレッド・ライブラリーでのみ非同期メッセージ処理をサポートします。そのため、ここでは非ブロッキング集合操作のパフォーマンスを最大限に引き出すようにマルチスレッド・ライブラリーを使用しています。インテル® MPI ライブラリーで非同期処理を有効にするには、**MPICH_ASYNC_PROGRESS** 環境変数を 1 に設定してください。

ここでは、非ブロッキング集合操作のオーバーラップを測定する **IMB-NBC** ベンチマークの結果と、**IMB-MPI1** で得られる通常の集合操作の測定結果を比較します。ブロッキング集合操作を実行してから計算を実行するのと、非ブロッキング集合操作により同量の計算とオーバーラップさせるのでは、どちらのほうが効率的かを検証します。グラフには非ブロッキング操作と CPU アクティビティーを同時に実行した場合 (IMB-NBC 出力の **time_ovrlp** 値) と、通信と計算を別々に行ってその時間を合計した場合の 2 つの測定時間を示します。2 つ目の値は、ブロッキング集合操作の時間 (**IMB-MPI1** ベンチマークで測定) とテスト CPU アクティビティーの時間 (**IMB-NBC** 出力の **time_CPU** 値) の合計です。

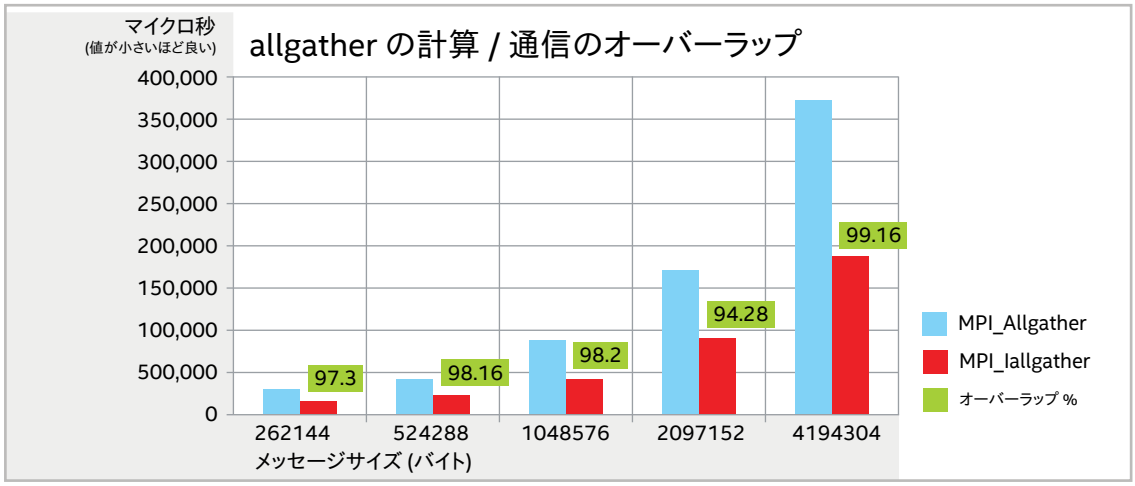
グラフに示す値は、4 つの IVT ノードで 48 プロセス (ノードあたり 12 プロセス) を使用した結果です。各ノードには、2 つのインテル® Xeon® プロセッサー E5-2697 と 1 つの Mellanox* Connectx*-3 InfiniBand* アダプターが搭載されています。

ialltoall および iallgather ベンチマークから得られるオーバーラップは理想に近いものです。図 1 と図 2 は、代表的な結果としてメッセージサイズの大きなもののみを示しています。非同期処理を行う非ブロッキング操作ではオーバーヘッドが大幅に増えるにもかかわらず (これは特に小さなメッセージサイズで顕著です)、通常のブロッキング集合操作と計算の組み合わせよりも効率的であることが分かります。しかし、常にそうなるとは限りません。図 3 は、小さなメッセージサイズのブロードキャスト・ベンチマークの結果です。オーバーラップの割合は比較的良いものの (約 60 ～ 70%)、通常のブロッキング・ブロードキャスト操作のほうが高速で、メッセージサイズが 2KB 以下の場合はパフォーマンス効率も良くなります。

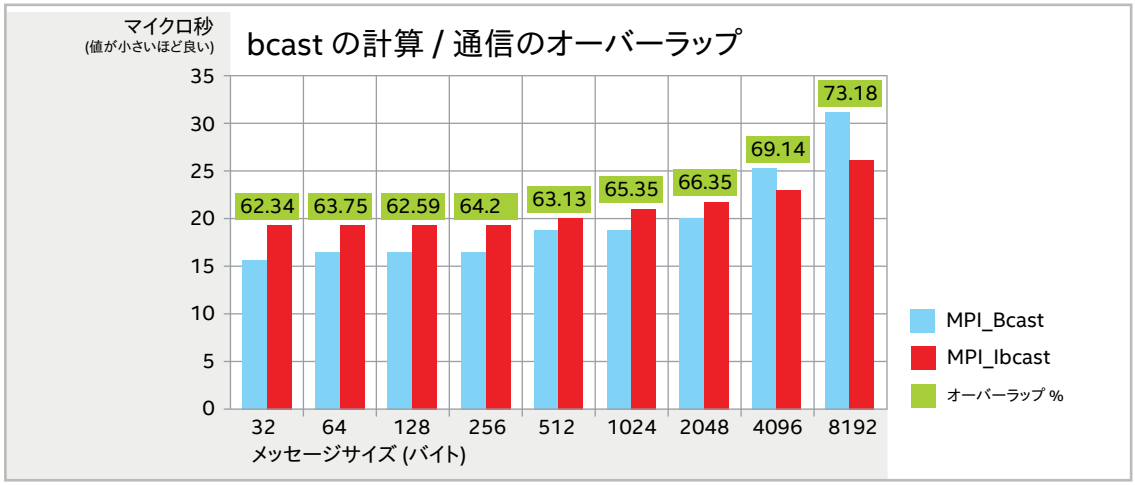
インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



1 MPI_Ialltoall と計算をオーバーラップさせた場合と MPI_Alltoall を実行してから計算を行った場合



2 MPI_Iallgather と計算をオーバーラップさせた場合と MPI_Allgather を実行してから計算を行った場合



3 MPI_Ibcast と計算をオーバーラップさせた場合と MPI_Bcast を実行してから計算を行った場合

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

これらの結果から次のことが分かります。

- 非ブロッキング集合操作は計算とオーバーラップすることで、特にメッセージサイズが中～大の場合、ブロッキング操作と比べて大幅なパフォーマンス向上をもたらします。
- 場合によっては、非ブロッキング操作と非同期メッセージ処理によって生じるオーバーヘッドがオーバーラップの利点を打ち消してしまうことがあります。これは、通常レイテンシーが重要になる小さなメッセージサイズで起こる傾向にあります。

新しい RMA インターフェイス

MPI-3 標準はさまざまなメモリーモデル、新しい通信と同期呼び出し、RMA ウィンドウの新しい作成方法を含むいくつかの主要機能の追加により、RMA インターフェイスを大幅に拡張します。異なるメモリーモデルのサポートは最も重要な新機能の 1 つで、キャッシュ・コヒーレンシーがサポートされる (ユニファイド・モデル) 場合とされない場合 (MPI-2 標準に特有の個別モデル) を区別します。この区別により、ハイパフォーマンスな MPI 実装が可能になるとともに、プログラミングが容易になります。同期セマンティクスの主な更新は、パッシブターゲット通信モードを現在の HPC のニーズに合わせるためのものです。MPI-3 標準で改善が試みられている以前の RMA インターフェイスの主な問題は、http://upc.lbl.gov/publications/bonachea-duell-mpi.pdf **(英語)** で説明されています。

ベータ版 IMB 4.0 以前のバージョンでは、アクティブ通信モードのみを用いる RMA ベンチマークのセットが提供されていました。これらのベンチマークは、`IMB-EXT` モジュールにあります。ベータ版 IMB 4.0 は、MPI-3 の新しい RMA 機能とパッシブターゲット通信モードに注力しています。新しい IMB-RMA モジュールには 19 のベンチマークが含まれており、新しいアトミック関数、`MPI_Put()` および `MPI_Get()` 操作を使用するハロゲン交換、異なるターゲットへの同時 RMA 呼び出しを測定します。これらのベンチマークはすべて、パッシブターゲット通信モードを利用します。

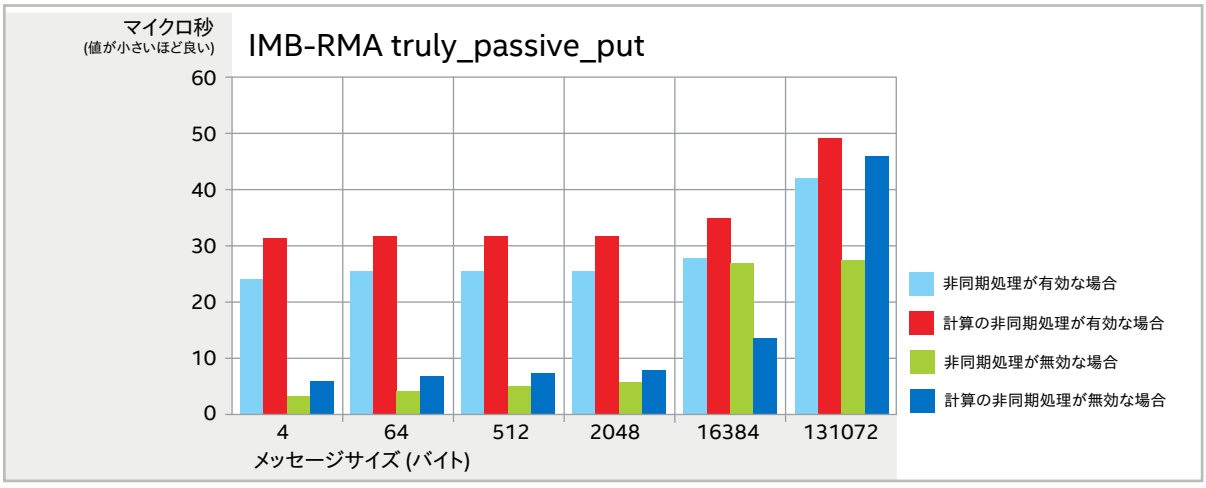
新しい RMA ベンチマークの 1 つは、MPI 実装が通信の自然なパッシブモードをサポートするかどうかを示します。自然なパッシブモードでは、発信元のプロセスがアクセスエポックを完了できるように、ターゲットプロセスで MPI を呼び出す必要はありません。これは、前述の非同期メッセージ処理によって、あるいはハードウェアの通信オフロード機能によってサポートされる可能性があります。ベータ版インテル® MPI ライブラリー 5.0 の RMA 操作は、通常のポイントツーポイント・メッセージを用いて実装されているため、確実に完了するようにターゲットと発信元で処理される必要があります。前述したように、非同期メッセージ処理はインテル® MPI ライブラリーのマルチスレッド・バージョンでサポートされています。シングルスレッド・バージョンではパッシブ・ターゲット・モードはサポートされていません。そのため、ターゲットは MPI を呼び出して処理しなければなりません。ここでは、非同期処理を行った場合とシングルスレッド・バージョンを利用した場合の結果を比較し、インテル® MPI ライブラリーのパッシブ・モード・サポートの利点を評価します。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

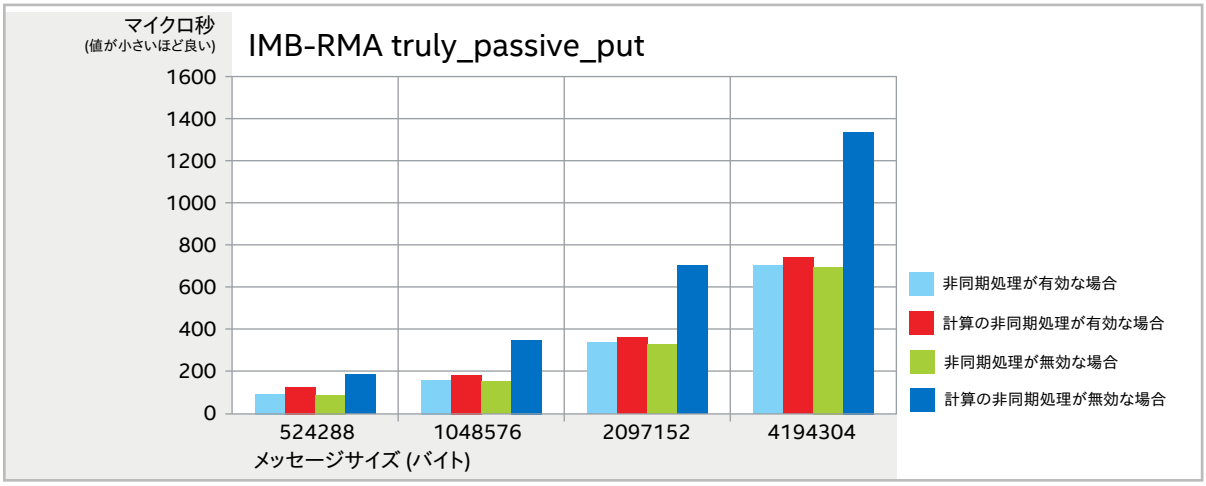
上記のベンチマークは次の 2 つの時間を測定します。

1. ターゲットプロセスが (確実に処理されるように) `MPI_Barrier()` 呼び出しを実行している間、発信元プロセスが `MPI_Put()` 操作の完了に費やした時間
2. ターゲットプロセスが MPI スタック外で計算を実行し `MPI_Barrier()` を呼び出す間、発信元プロセスが `MPI_Put()` 操作の完了に費やした時間。計算時間は、発信元プロセスがステップ 1 を完了するのに必要な時間とほぼ同じになるようにします。そのためには、`MPI_Put()` の完了に必要な時間の値を発信元プロセスからターゲットプロセスへ送ります。

ターゲットプロセスと発信元プロセスは、各 `MPI_Put()` 操作の前に同期されます。そのため、直接通信オフロード機能がない場合、あるいはメッセージ処理を実行するスレッドが異なる場合、ステップ 2 の時間はステップ 1 の時間よりも大きく (約 2 倍に) なるでしょう。ただし、MPI 実装が RMA でパッシブモードを提供している場合、これらの時間はほぼ同じになります。



4 非同期処理のパッシブ・ベンチマーク結果への影響 (小～中程度のメッセージサイズ)



5 非同期処理のパッシブ・ベンチマーク結果への影響 (大きなメッセージサイズ)

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

図 4 と**図 5** は、非同期メッセージ処理によりオーバーヘッドが著しく増加することを示しており、これはメッセージサイズが 128KB 以下の場合に顕著です。しかし、ラージ・メッセージサイズではオーバーヘッドの影響があまり見受けられず、ターゲットの計算パフォーマンスによって発信元の `MPI_Put()` 操作の完了が遅れることはありません。

ベータ版インテル® MPI ライブラリー 5.0 でパッシブモードの通信モデルの動作を分析した結果、このライブラリーでは非同期処理でパッシブモードがサポートされているものの、改善の余地があることが分かりました。これには、RDMA 処理のネイティブサポートやさまざまなパフォーマンスの最適化 (例えば、オーバーヘッドが大きく、マルチスレッド・バージョンでのみ利用可能な非同期メッセージ処理の改善) などがあります。MPI-3 の新しい RMA インターフェイスは柔軟な一方向操作のインターフェイスを提供します。その利点については、**http://htor.inf.ethz.ch/publications/img/mpi_mpi_hybrid_programming.pdf (英語)** を参照してください。これらの利点により、MPI 実装のパフォーマンス効率が向上し、さまざまな PGAS 言語に対する競争力も上がるでしょう。

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

まとめ

MPI-3 標準は、MPI ユーザーがパフォーマンス効率の良いプログラムを記述できるように柔軟なインターフェイスを提供します。ここでは、ベータ版インテル® MPI ライブラリー 5.0 を例に 2 つの重要な機能、非ブロッキング集合操作と新しい RMA インターフェイスについて述べました。これらの機能によって MPI アプリケーションのパフォーマンスを向上できるでしょう。既存の MPI プログラムでは、新しい MPI-3 機能を利用するのにアプリケーションの変更が必要になるかもしれませんが、それだけの価値があることはここで示したように明らかです。

参考文献

1. Message Progression in Parallel Computing—To Thread or Not to Thread, Torsten Hoefler and Andrew Lumsdaine.
   http://htor.inf.ethz.ch/publications/index.php?pub=75 (英語)
2. A Message-Passing Interface Standard 3.0,
   http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (英語)
3. ベータ版インテル® MPI ライブラリー 5.0 Readme:
   http://software.intel.com/en-us/articles/intel-mpi-library-50-beta-readme (英語)

インテル® MPI ライブラリーを試そう >

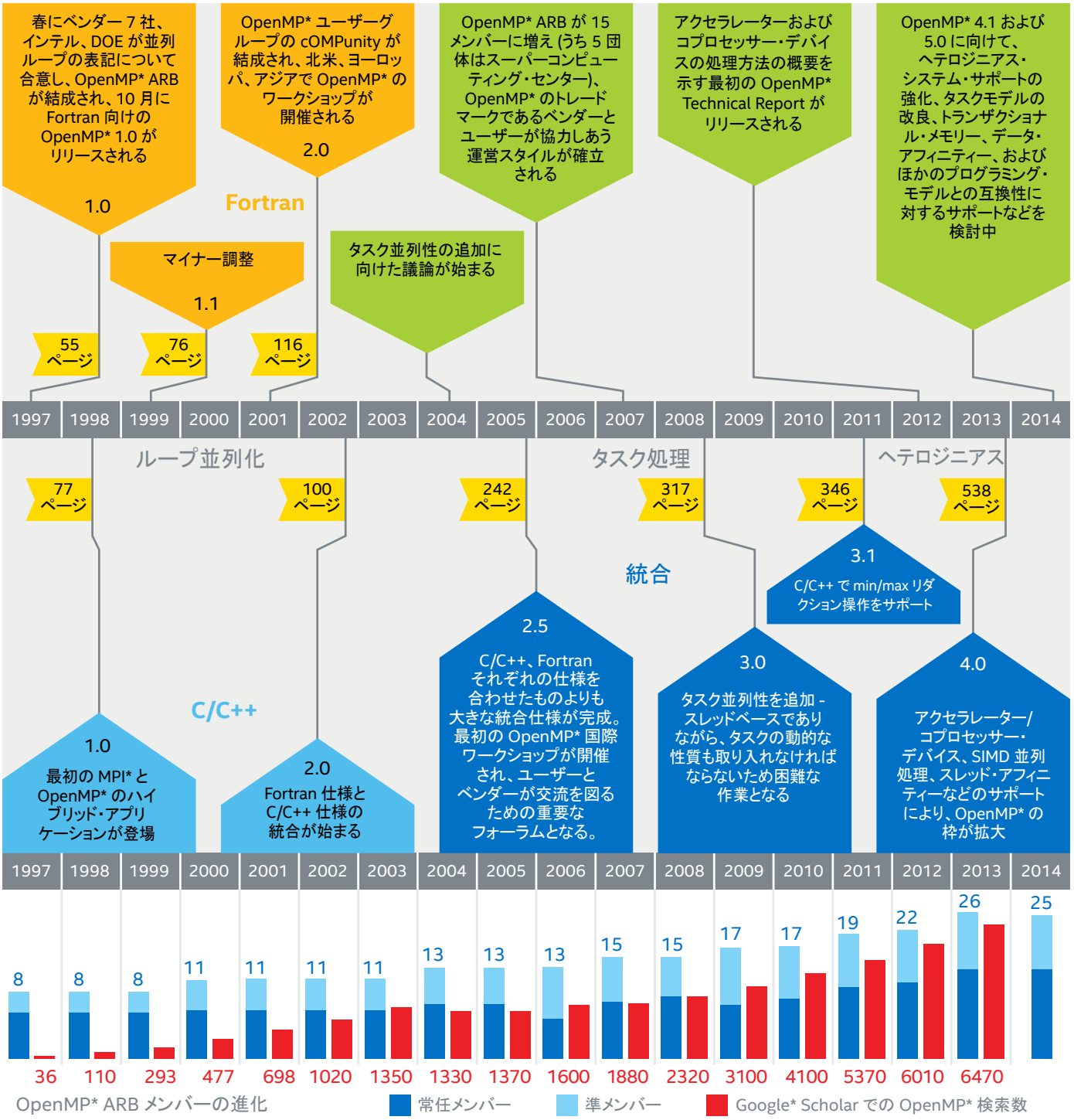
インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

OpenMP* 年表

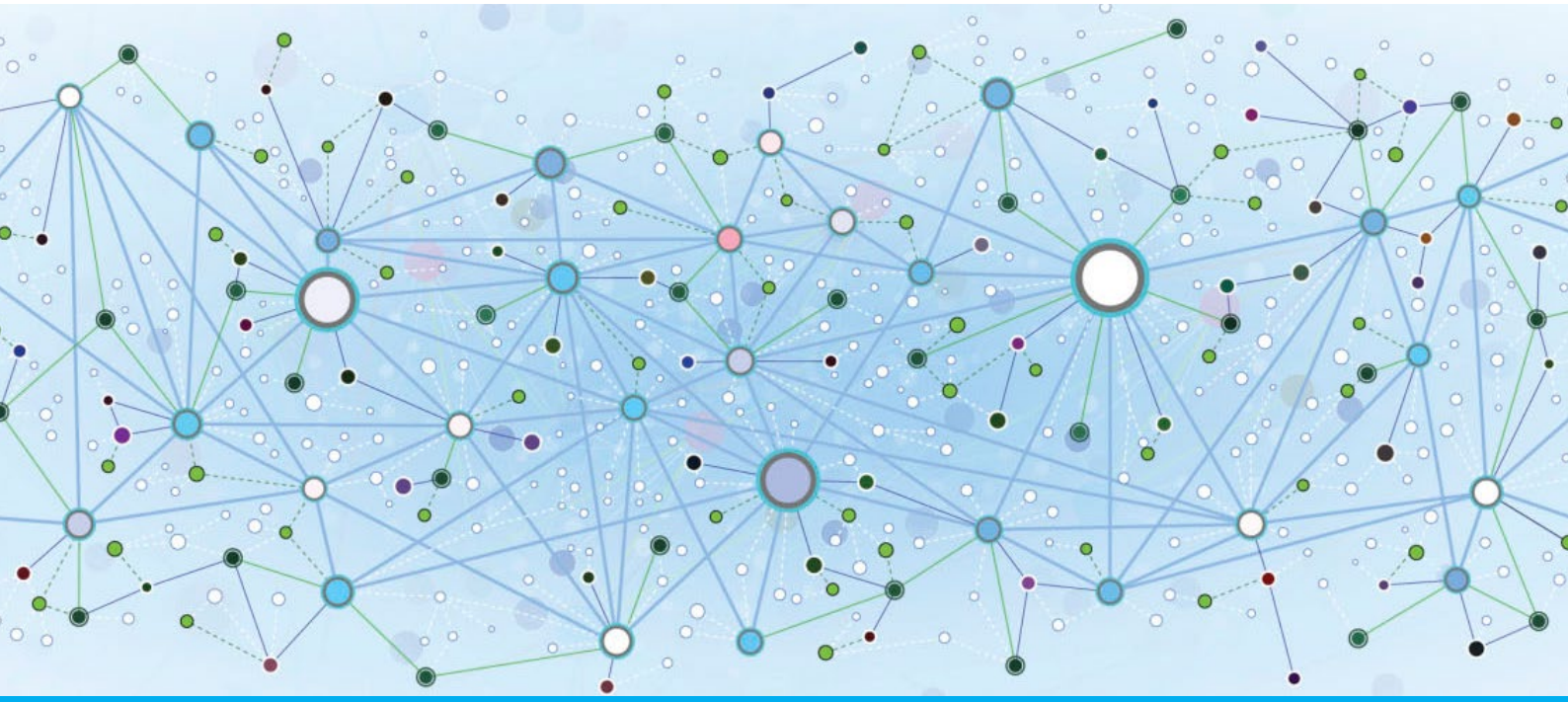
Jim Cownie OpenMP* アーキテクト、**Alejandro Duran** アプリケーション・エンジニア、**Michael Klemm** シニア・アプリケーション・エンジニア、**Luke Lin OpenMP*** ソフトウェア製品マネージャー

1996 ベンダーごとにループの並列処理に対するソリューションがやや異なるため、移植性と保守性において問題が生じる

Kuck and Associates, Inc. (KAI) | SGI | Cray | IBM | High Performance Fotran (HPF) | Parallel Computing Forum (PCF)



インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



インテル® アーキテクチャーにおける OpenCL* 投資の活用

James Cownie インテル コーポレーション 主任エンジニア
Simon McIntosh-Smith ブリストル大学
Arik Narkis インテル コーポレーション OpenCL* パフォーマンス・アーキテクト
Ayal Zaks インテル コーポレーション OpenCL* MIC コンパイラー・マネージャー

OpenCL* (Open Computing Language) は、クロスプラットフォームの並列プログラミング向けロイヤルティー・フリーの標準仕様です。OpenCL* プログラミング・モデルは、汎用 CPU、グラフィックス・プロセッサー、コプロセッサーを含む多種多様なデバイスから成るヘテロジニアス・プラットフォームをサポートします。OpenCL* 仕様には、ローカルメモリー、バリア、イメージ、サンプラー、ワークグループなど、ほかの仕様にはない GPU 用の低レベルの機能が含まれています。インテルは、2010 年に汎用 CPU で OpenCL* をサポートし、2013 年には高度な並列性を備えた x86 コプロセッサーであるインテル® Xeon Phi™ コプロセッサーでもサポートしました。

OpenCL* の魅力は、デバイス間の移植性が保証されること、そしてハードウェア・ベンダーにより広くサポートされていることです。通常、特定のターゲット GPU 向けに設計された OpenCL* コードは、x86 でシームレスに、あるいはコードをわずかに変更するだけで動作します。しかし、さまざまなデバイスに渡ってパフォーマンスの可搬性が保証されるわけではありません。特に、GPU 向けにチューニングされている OpenCL* コードはほとんどの場合、インテル® Xeon Phi™ コプロセッサーで最良のパフォーマンスを発揮できません。逆も同様です。これは一般に、ローカルメモリーやバリアなどの GPU ハードウェア固有の機能を使用していたり、ワークグルー

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

プが少なすぎたり (並列処理の粒度が粗すぎる場合)、あるいはワークグループが小さすぎる (並列処理の粒度が細かすぎる)ことが原因です。

代表的なインテル® Xeon Phi™ コプロセッサーには 60 個のインテル® アーキテクチャー・コアが搭載されており、コアごとに最大 4 つのハードウェア・スレッドを実行できます。OpenCL* ランタイムは、この 240 のハードウェア・スレッドをワークグループ全体の粒度でスケジュールします。そのため、ワークグループが 240 未満の OpenCL* タスク (NDRange) では実行リソースが十分に活用されません。リソースの利用率を高めて効率良い負荷バランスを実現するには、NDRange ごとに少なくとも 1000 ワークグループを目標にするべきでしょう。

インテル® Xeon Phi™ コプロセッサー上で OpenCL* を利用すると、細粒度の NDRanges ではランタイム・スケジューリング・オーバーヘッドが生じる可能性があります。このオーバーヘッドは各ワークグループの実行時間が 200 マイクロ秒未満の場合、パフォーマンスに影響します。そのため、粗粒度のワークグループにしたほうが良いでしょう。例えば、複数のカーネルを 1 つの大きなカーネルにまとめてワークグループの実行時間を増やします。また、別の方法としては、ワークグループが前述の最小数を満たしつつ、より大きくかつ少なくなるように NDRange を設定します。

インテルの OpenCL* コンパイラーは、NDRange の次元 0 に基づいて (その値を把握していないにもかかわらず)、各ワークグループ・ルーチンを暗黙にベクトル化します。その結果、次元 0 のループは、ターゲットのベクトルサイズでアンロールされます。例えば、生成される 3 次元のカーネルコードは次のようになります。

```
1   _kernel kernel_name(/* カーネル引数 */)
2    {
3      for (int dim2=0; dim2 < get_local_size(2); dim2++)
4        for (int dim1=0; dim1 < get_local_size(1); dim1++)
5        {
6          for (int dim0=0; dim0 + VECTOR_SIZE < get_local_size(0);
              dim0 += VECTOR_SIZE)
7           /* Vectorized_Kernel_Body */
8          for (; dim0 < get_local_size(0); dim0++)
9           /* Original_Kernel_Body -  リマインダー・ループ */
10      }
11  }
```

1 ワークグループの次元 0 でベクトル化されるカーネル

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

インテル® Xeon Phi™ コプロセッサーでは、使用されるデータ型に関係なく、ベクトルサイズは 16 に設定されます。そのため、次元 0 のローカルサイズは 16 の倍数にしたほうが良いでしょう。さらに、次元 0 でユニットストライドによりバッファーへアクセスすると、より効率的なベクトルコードになります。

GPU 向けにチューニングされている既存の OpenCL* コードは、そのままではインテル® Xeon Phi™ コプロセッサーで最適なパフォーマンスが得られない可能性があります。しかし、適度なチューニングにより、インテル® Xeon Phi™ コプロセッサーと GPU のどちらでも優れたパフォーマンスを達成できます。その一例として、ブリストル大学の Simon McIntosh-Smith 氏らによって開発された BUDE 分子ドッキングコードが挙げられます。BUDE の OpenCL* 実装は、インテル® Xeon Phi™ コプロセッサーを含むさまざまなメニーコア・アーキテクチャーで、ピーク時浮動小数点パフォーマンスの 32% 以上を維持します。OpenCL* によりパフォーマンスの可搬性を実現した BUDE のアプローチについては、最近の論文「High Performance in silico Virtual Drug Screening on Many-Core Processors」[1] (英語) で説明されています。

ブリストル大学は、2014 年 2 月からインテル® Parallel Computing Center Program に参加し、OpenCL* および OpenMP* でインテル® Xeon Phi™ コプロセッサーを活用する方法を調査しています。[2] ブリストル大学がインテル® Xeon Phi™ コプロセッサー向けに最適化を行ったコードには、BUDE 分子ドッキングコード、ROTORSIM CFD コード [3]、Mantevo ベンチマーク・スイートの CloverLeaf ミニアプリ [4] などがあります。

まとめ

OpenCL* は、2008 年の終わりに最初の仕様 (1.0) がリリースされて以来、大きく進化しました。現在の実装は、さまざまなヘテロジニアス・プラットフォームにわたって移植可能なプログラミングのデファクト・スタンダードを提供し、開発投資を活かすことができます。OpenCL* の仕様と実装はどちらも進化し続けています。パフォーマンスの可搬性は、研究機関や業界において多くの OpenCL* チームが取り組んでいる大きな課題です。ここでは、インテル® Xeon Phi™ コプロセッサーに関するパフォーマンスの可搬性について、考慮すべき点を概説しました。

OpenCL* チュートリアル、インテル® Xeon® プロセッサーおよびインテル® Xeon Phi™ コプロセッサー向け最適化ガイドなどの関連情報は、**http://www.isus.jp/article/idz/opencl-sdk/** を参照してください。

インテル® SDK for
OpenCL* Applications を試そう >

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

参考文献

1. High Performance in silico Virtual Drug Screening on Many-Core Processors, McIntosh-Smith, S.N., Price, J.R., Sessions, R.B., and Ibarra, A.A., International Journal of High Performance Computing Applications, 2014.
2. http://www.bristol.ac.uk/news/2014/january/10099.html (英語)
3. On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures, McIntosh-Smith, S.N., Boulton, M., Curran, D., and Price, J.R., appearing in the Proceedings of the 29th International Supercomputing Conference, ISC 2014, Leipzig, Germany, June 23–27, 2014.
4. Using Mini Applications in a Mantevo Framework for Optimizing Sandia’s SPARC CFD Code on Multicore, Many-Core, and GPU-Accelerated Compute Platforms, Barnette, D.W., Barrett, R.F., Hammond, S.D., Jayaraj, J., and LarosIII, J.H., Technical Report, Sandia National Laboratories, 2012.

最適化に関する注意事項

インテル® コンパイラーは、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® マイクロアーキテクチャーに非固有の特定の最適化は、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。この注意事項の適用対象である特定の命令セットの詳細は、該当する製品のユーザー・リファレンス・ガイドを参照してください。

改訂 #20110804

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。

BLOG HIGHLIGHTS

トランザクショナル・メモリー・サポート: speculative_spin_rw_mutex (コミュニティー・プレビュー機能)

CHRISTOPHER HUS »

以前の投稿で最新世代のプロセッサーで利用可能なインテル® トランザクショナル・シンクロナイゼーション・エクステンション (インテル® TSX) について述べ、HLE インターフェイス (**speculative_spin_mutex**) の 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) 実装を紹介しました。この記事では、インテル® TBB 4.2 Update 2 のプレビュー機能である **speculative_spin_rtw_mutex** について説明します。

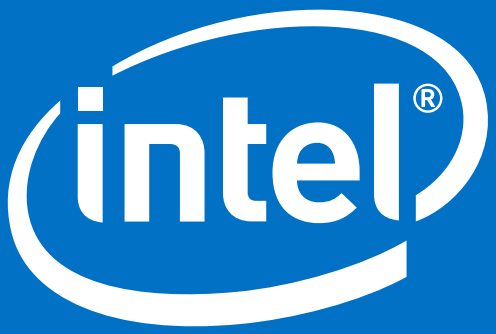
speculative_spin_rtw_mutex は排他制御に RTM を使用し、同時読み取りと同時書き込みの両方を許可します。スペキュレーションなしで操作を実行しなければならない場合に備えて、**spin_rw_mutex** も用意されています。

mutex で保護されたコードの同時実行が競合しない場合、明示的にロックを取得せずにすべての読み取りは完了し、書き込みはアトミックにコミットされます。

競合やスペキュレーティブ実行を妨げる別の問題によってトランザクションがアボートされると、**speculative_spin_rtw_mutex** はトランザクションの再実行を試みるか、実際にロックを取得します。書き込み操作がロックを取得する場合、すべての読み取りと書き込みはアボートされ、書き込みが完了するのを待機します。この場合、書き込みが完了した時点で、待機していたトランザクションはリタイアしている可能性があります。読み取り操作がロックを取得する場合、すべてのスペキュレーティブな書き込みはアボートされ、読み取り操作がロックを解放するのを待機します。この場合、ロックが解放された時点で、待機していたトランザクションはリタイアしている可能性があります。実際の読み取りとスペキュレーティブな読み取りは並列に行えます。この一連の処理はすべて、インテル® TBB 実装の一部として内部で行われます。

この記事の続きはこちらでご覧になれます。 ›

インテル® ソフトウェア製品のパフォーマンスおよび最適化に関する注意事項については、
http://software.intel.com/en-us/articles/optimization-notice/#opt-jp を参照してください。



Software

The Parallel Universe

© 2014 Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Xeon、
Xeon Phi、Cilk、VTune は、アメリカ合衆国および / または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。
OpenCL および OpenCL ロゴは、Apple Inc. の商標であり、Khronos の使用許諾を受けて使用しています。