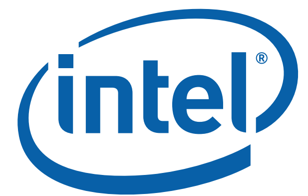


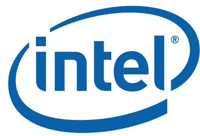
インテル® マス・カーネル・ライブラリー Linux* 版

ユーザーズガイド

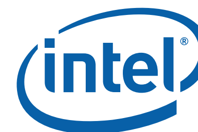
2007 年 10 月

資料番号 : 314774-005JA

Web サイト :
http://developer.intel.com (英語)
http://www.intel.com/jp/developer/software/products/ (日本語)



バージョン	バージョン情報	日付
-001	初版。インテル ® マス・カーネル・ライブラリー (インテル ® MKL) 9.0 gold リリースについて説明。	2006 年 9 月
-002	インテル ® MKL 9.1 beta リリースについて説明。「はじめに」、「LINPACK ベンチマークと MP LINPACK ベンチマーク」の章と「サードパーティー・インターフェイスのサポート」の付録を追加。既存の章を拡張。ドキュメントを再構成。例のリストを追加。	2007 年 1 月
-003	インテル ® MKL 9.1 gold リリースについて説明。既存の章を拡張。ドキュメントを再構成。ILP64 インターフェイスに関する他の内容を追加。「インテル ® MKL を Eclipse CDT でリンクする場合の構成」セクションを第 3 章に追加。クラスターに関する内容を 1 つの独立した第 9 章「インテル ® マス・カーネル・ライブラリー・クラスター・ソフトウェアの使用」に移動して再構成し、適切なリンクを追加。	2007 年 6 月
-004	インテル ® MKL 10.0 Beta リリースについて説明。レイヤーモデルについての記述を第 3 章に追加し、本書の内容をモデルに合わせて調整。スタートアップ時の環境変数の自動設定についての記述を第 4 章に追加。新しいインテル ® MKL スレッド化コントロールについての記述を第 6 章に追加。インテル ® MKL のユーザーズガイドとインテル ® MKL クラスター・エディションのユーザーズガイドを併合し、それぞれの製品の統合を反映。	2007 年 9 月
-005	インテル ® MKL 10.0 Gold リリースについて説明。インテル ® MKL を Eclipse CDT 4.0 でリンクする場合の説明を第 3 章に追加。インテル ® 互換 OpenMP* ランタイム・コンパイラー・ライブラリー (`libiomp`) について記述。	2007 年 10 月



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

インテルによる書面での同意がない限り、インテル製品は、インテル製品の停止を起因とする人身傷害または死亡を想定して設計されていません。

インテル製品は、予告なく仕様や説明が変更される場合があります。機能または命令の一覧で「留保」または「未定義」と記されているものがありますが、その「機能が存在しない」あるいは「性質が留保付である」という状態を設計の前提にしないでください。これらの項目は、インテルが将来のために留保しているものです。インテルが将来これらの項目を定義したことにより、衝突が生じたり互換性が失われたりしても、インテルは一切責任を負いません。この情報は予告なく変更されることがあります。この情報だけに基づいて設計を最終的なものとしないでください。

本資料で説明されている製品には、エラッタと呼ばれる設計上の不具合が含まれている可能性があり、公表されている仕様とは異なる動作をする場合があります。現在確認済みのエラッタについては、インテルまでお問い合わせください。

最新の仕様をご希望の場合や製品をご注文の場合は、お近くのインテルの営業所または販売代理店にお問い合わせください。

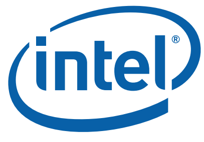
本書で紹介されている注文番号付きのドキュメントや、インテルのその他の資料を入手するには、1-800-548-4725 (アメリカ合衆国) までご連絡いただくか、インテルの Web サイトを参照してください。

インテル・プロセッサー・ナンバーはパフォーマンスの指標ではありません。プロセッサー・ナンバーは同一プロセッサー・ファミリー内の製品の機能を区別します。異なるプロセッサー・ファミリー間の機能の区別には用いません。詳細については、http://www.intel.co.jp/jp/products/processor_number/ を参照してください。

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

* その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

© 2006 - 2007 Intel Corporation. 無断での引用、転載を禁じます。



目次

第 1 章　概要
テクニカルサポート 1-1
本書について 1-1
目的 1-2
対象読者 1-2
本書の構成 1-2
表記規則 1-3

第 2 章　はじめに
インストールの確認 2-1
バージョン情報の取得 2-1
サポートするコンパイラー 2-2
インテル ® MKL の使用を開始する前に 2-2

第 3 章　インテル ® マス・カーネル・ライブラリーの構造
高レベル・ディレクトリー構造 3-1
レイヤーモデルの概念 3-2
レイヤー 3-3
ライブラリーの逐次バージョン 3-4
ILP64 プログラミングのサポート 3-4
インテル ® MKL のバージョン 3-10
詳細なディレクトリー構造 3-10
ダミー・ライブラリー 3-18
ドキュメント・ディレクトリーの内容 3-19

第 4 章　開発環境の構成
環境変数の設定 4-1

プロセスの自動化 4-1
インテル ® MKL を Eclipse CDT でリンクする場合の構成........ 4-2
Eclipse CDT 4.0 の構成 4-2
Eclipse CDT 3.x の構成 4-3
構成ファイルを使用したライブラリーのカスタマイズ........ 4-4

第 5 章 アプリケーションとインテル ® マス・カーネル・ライブラリーのリンク
リンクモデルの違い 5-1
スタティック・リンク 5-1
ダイナミック・リンク 5-1
リンクモデルの選択........ 5-2
インテル ® MKL 固有リンクの推奨 5-2
リンクコマンドの構文 5-3
リンクするライブラリーの選択 5-5
スレッド化ライブラリーのリンク........ 5-6
その他のリンクの例........ 5-7
リンクにおける注意........ 5-9
カスタム共有オブジェクトの構築........ 5-10
インテル ® MKL カスタム共有オブジェクト・ビルダー........ 5-10
メイクファイル・パラメーターの指定........ 5-10
関数のリストの指定........ 5-11

第 6 章
パフォーマンスとメモリーの管理
インテル ® MKL 並列処理の使用 6-1
スレッド数を設定する手法 6-2
実行環境における競合の回避........ 6-2
OpenMP 環境変数を使用したスレッド数の設定 6-3
ランタイムのスレッド数の変更 6-4
新しいスレッド化コントロールの使用........ 6-6
パフォーマンスを向上するためのヒントと手法 6-10
コーディング手法 6-10
ハードウェア構成のヒント 6-11
マルチコア・パフォーマンスの管理 6-12
非正規化数の演算 6-12
FFT 最適化基数 6-13
インテル ® MKL メモリー管理の使用 6-13
メモリー関数の再定義 6-14

第 7 章 言語固有の使用法オプション
言語固有インターフェイスとインテル ® MKL の使用 7-1
混在言語プログラミングとインテル ® MKL 7-4
LAPACK、BLAS、および CBLAS ルーチンの C 言語環境からの呼び出し 7-4
C/C++ コードで複素数を返す BLAS 関数の呼び出し 7-5
インテル ® MKL 関数の Java アプリケーションからの呼び出し 7-8

第 8 章 コーディングのヒント
数値計算安定性のためのデータの整列 8-1

第 9 章 インテル ® マス・カーネル・ライブラリー・クラスター・ソフトウェアの使用
ScaLAPACK および クラスター FFT とのリンク 9-1
スレッド数の設定 9-2
共有ライブラリーの使用 9-2
ScaLAPACK テスト 9-3
ScaLAPACK およびクラスター FFT とのリンクの例 9-3
C モジュールの例 9-3
Fortran モジュールの例 9-4

第 10 章 LINPACK ベンチマークと MP LINPACK ベンチマーク
Intel® Optimized LINPACK Benchmark for Linux 10-1
内容 10-1
ソフトウェアの実行 10-2
既知の制限事項 10-3
Intel® Optimized MP LINPACK Benchmark for Clusters 10-3
内容 10-4
MP LINPACK の構築 10-5
新機能 10-6
クラスターのベンチマーク 10-6

付録 A インテル ® マス・カーネル・ライブラリー言語インターフェイスのサポート

付録 B サードパーティー・インターフェイスのサポート
GMP* 関数 B-1
FFTW インターフェイスのサポート B-1

索引

表

表 1-1 表記規則 1-3
表 2-1 開始前に知っておく必要がある項目 2-2
表 3-1 高レベル・ディレクトリー構造 3-1
表 3-2 インテル ® MKL の ILP64 の概念 3-6
表 3-3 ILP64 インターフェイス用のコンパイラー・オプション 3-7
表 3-4 整数型 3-8
表 3-5 インテル ® MKL インクルード・ファイル 3-8
表 3-6 インテル ® MKL の ILP64 サポート 3-9
表 3-7 詳細なディレクトリー構造 3-10
表 3-8 doc ディレクトリーの内容 3-19
表 5-1 インテル ® MKL リンクモデルの比較 5-2
表 5-2 スレッド化レイヤーと RTL レイヤーの選択 5-7
表 6-1 スレッド化モデル別の実行環境における競合の回避方法 6-3
表 6-2 スレッド化コントロール用のインテル ® MKL 環境変数 6-7
表 6-3 MKL_DOMAIN_NUM_THREADS の値の解釈 6-9
表 7-1 インターフェイス・ライブラリーとモジュール 7-1
表 10-1 LINPACK Benchmark の内容 10-1
表 10-2 MP LINPACK Benchmark の内容 10-4

例

例 4-1 インテル ® MKL 構成ファイル 4-4
例 4-2 構成ファイルを使用したライブラリー名の再定義 4-5
例 6-1 スレッド化用のプロセッサー数の変更 6-4
例 6-2 スレッド数を 1 に設定 6-7
例 6-3 オペレーティング・システムでアフィニティー・マスクを設定してインテル ® コンパイラーを使用 6-12
例 6-4 メモリー関数の再定義 6-14
例 7-1 複素 BLAS レベル 1 関数の C からの呼び出し 7-6
例 7-2 複素 BLAS レベル 1 関数の C++ からの呼び出し 7-7
例 7-3 BLAS を C から直接呼び出す代わりに CBLAS インターフェイスを使用 ... 7-8
例 8-1 16 バイト境界でアドレスをアライメント 8-2

1 概要

インテル ® マス・カーネル・ライブラリー (インテル ® MKL) は、最大限のパフォーマンスを要求する科学、エンジニアリング、金融アプリケーション用に高度に最適化された、スレッドセーフな数値演算ルーチンを提供します。

テクニカルサポート

インテルでは、基本操作のヒント、製品に関する確認済みの問題点、製品のエラッタ、ライセンス情報、ユーザーフォーラムなど、豊富なセルフヘルプ情報を利用できるサポート Web サイトを提供しています。詳細は、インテル ® MKL サポート Web サイト
(http://www.intel.com/software/products/support/) を参照してください。

本書について

インテル ® MKL を使用したアプリケーションの開発に成功するには、基本的に 2 種類の情報、リファレンス情報と使用方法についての情報が必要です。*リファレンス情報*には、ルーチンの機能、パラメーターの説明、インターフェイス、呼び出し構文と戻り値が含まれます。この情報を入手するには、*『インテル ® MKL リファレンス・マニュアル』*を最初に参照してください。しかし、アプリケーションからインテル ® MKL ルーチンを呼び出す場合、リファレンス・マニュアルではわからない多くの疑問が生じます。例えば、ライブラリーの構成、特定のプラットフォームや解いている問題用にインテル ® MKL を設定する方法、インテル ® MKL を使用してアプリケーションをコンパイル、リンクする方法などです。また、インテル ® MKL のスレッド化とメモリー管理機能を活用して最適なパフォーマンスを得る方法も理解する必要があります。ルーチン呼び出しの詳細な処理 (例えば、異なるプログラミング言語やコーディング言語間のルーチン呼び出しで渡すパラメーター) も、よくある質問の 1 つです。計算精度の評価方法と改善方法についての質問もあるでしょう。これらの質問や問題は、インテル ® MKL の*使用方法についての情報*で説明します。

本書では、Linux* 上で実行しているユーザーのアプリケーションからインテル ® MKL ルーチンを呼び出すために必要な使用方法を説明します。特定の OS に依存しない機能に加えて、
インテル ® MKL Linux 版の特定の機能も説明しています。

本書には、表 A-1 (付録 A) でリストされている関数領域に含まれるインテル ® MKL ルーチンと関数の使用法が含まれています。

本書は、インテル ® MKL のインストールが完了した後に使用してください。製品のインストールがまだ完了していない場合、*『インテル ® MKL インストール・ガイド』* (`Install.txt`) を参照して
インストールを完了してください。

アプリケーションでライブラリーを使用する方法を参照するには、最新バージョンの*インテル® マス・カーネル・ライブラリー Linux 版リリースノート*とユーザーズガイドを組み合わせて使用してください。

目的

インテル® マス・カーネル・ライブラリー Linux 版ユーザーズガイドは、Linux 上でインテル® MKL の使用法の習得を助けることを目的に記述されています。特に、以下の項目について説明します。

- ライブラリーの使用を開始する前に製品のインストール後に実行する必要のある手順を説明します。
- ライブラリーを使用するためにライブラリーと開発環境を構成する方法を説明します。
- ライブラリーの構造を紹介します。
- アプリケーションとライブラリーをリンクする方法の詳細を説明し、簡単な使用法の例を紹介します。
- インテル® MKL Linux 版を使用してアプリケーションを作成、コンパイル、および実行する方法を詳細に説明します。

対象読者

本書は、ソフトウェア開発の初心者から熟練者までを含む、Linux プログラマーを対象としています。

本書の構成

本書は、以下の章および付録から構成されています。

第 1 章	「概要」。インテル® MKL の使用法の概念を紹介します。また、本書の目的と構成、表記規則について説明します。
第 2 章	「はじめに」。インストール後にインテル® MKL を使用するために必要な手順と基本的な情報を説明します。
第 3 章	「インテル® マス・カーネル・ライブラリーの構造」。インストール後のインテル® MKL ディレクトリーの構造と、ライブラリーのバージョンおよび種類について説明します。
第 4 章	「開発環境の構成」。ライブラリーを使用するためにインテル® MKL と開発環境を構成する方法を説明します。
第 5 章	「アプリケーションと インテル® マス・カーネル・ライブラリーのリンク」。リンクモデル (スタティックとダイナミック) を比較します。インテル® MKL ライブラリーのリンクに使用される一般的なリンク行の構文を説明します。特定のプラットフォームや関数領域用にアプリケーションとリンクするライブラリーと、カスタム・ダイナミック・ライブラリーのビルド方法も説明します。

第 6 章	「パフォーマンスとメモリーの管理」。インテル ® MKL のスレッド化について説明し、ライブラリーのパフォーマンスを向上するためのコーディング・テクニックとハードウェア構成を示します。また、インテル ® MKL のメモリー管理機能の特徴について説明し、デフォルトで使用するライブラリーのメモリー関数を独自の関数と置換する方法を示します。
第 7 章	「言語固有の使用法オプション」。混在言語プログラミングと言語固有のインターフェイスの使用について説明します。
第 8 章	「コーディングのヒント」。特定の用途を満たすために役立つコーディングのヒントを示します。
第 9 章	「インテル ® マス・カーネル・ライブラリー・クラスター・ソフトウェアの使用」。ScaLAPACK とクラスター FFT の使用法について、C と Fortran 固有のリンク例を含む、関数領域を使用するアプリケーションのリンク方法を主に説明します。また、サポートしている MPI の情報も示します。
第 10 章	「LINPACK ベンチマークと MP LINPACK ベンチマーク」。Intel® Optimized LINPACK Benchmark for Linux および Intel® Optimized MP LINPACK Benchmark for Clusters について説明します。
付録 A	「インテル ® マス・カーネル・ライブラリー言語インターフェイスのサポート」。インテル ® MKL で各関数領域用に用意されている言語インターフェイスについての情報を要約します。
付録 B	「サードパーティー・インターフェイスのサポート」。インテル ® MKL でサポートされている特定のインターフェイスについて簡単に説明します。

本書の最後には、索引も含まれています。

表記規則

本書では、以下のフォント表記と記号が使用されています。

表 1-1　　表記規則

斜体	強調および文書名を示します。例: *『インテル ® MKL リファレンス・マニュアル』*を参照してください。
等幅小文字	ファイル名、ディレクトリー名およびパス名を示します。 例: `libmkl_core.a` , `/opt/intel/mkl/10.0.039`
等幅小文字 / 大文字	コマンドおよびコマンドライン・オプションを示します。 例: `icc myprog.c -L$MKLPATH -I$MKLINCLUDE -lmkl -lguide -lpthread ;` C/C++ コードの一部。 例 : a = new double [SIZE*SIZE];
等幅大文字	システム変数を示します。例: `$MKLPATH`

表 1-1　表記規則 (続き)

等幅斜体	説明しているパラメーターを示します。例 : *lda* (ルーチン・パラメーター)、*functions_list* (makefile パラメーター)、その他。 角括弧で囲まれている場合。識別子、式、文字列、記号、値のプレースホルダーを示します。例 : *<mkl ディレクトリー>*。プレースホルダーの代わりに、これらの項目の いずれかを用いてください。
[項目]	大括弧は、括弧で囲まれた項目がオプションであることを示します。
{ 項目 | 項目 }	中括弧は、括弧で囲まれた項目の中から 1 つを選択することを示します。項目は縦線 (|) で区切ります。

はじめに

2

本章は、インテル® マス・カーネル・ライブラリー (インテル® MKL) Linux* 版の使用を開始するために必要な基本的な情報と製品のインストール後に実行する必要のある手順を説明します。

インストールの確認

インテル® MKL のインストールが完了したら、ライブラリーのインストールと設定が適切に行われていることを確認します。

1. まず、インストール用に選択したディレクトリーが作成されていることを確認します。デフォルトのインストール・ディレクトリーは /opt/intel/mkl/10.0.xxx です (xxx はパッケージ番号です。
   例： /opt/intel/mkl/10.0.039)。
2. コンピューターにインテル® MKL の複数のバージョンをインストールしている場合、*使用する*バージョンを指定するようにビルドスクリプトを更新します。コンピューターにインテル® MKL のベータ版をインストールしている場合、ほかのバージョンをインストールする前に削除する必要があります。
3. 以下の 6 つのファイルが tools/environment ディレクトリーにインストールされます。

   mklvars32.sh

   mklvarsem64t.sh

   mklvars64.sh

   mklvars32.csh

   mklvarsem64t.csh

   mklvars64.csh

   現在のユーザーシェルで、これらのファイルを使用して INCLUDE、LD_LIBRARY_PATH、および MANPATH などの環境変数を設定します。

バージョン情報の取得

インテル® MKL では、ライブラリーに関する情報 (例えば、バージョン番号) を取得する方法を提供しています。この情報を取得する方法は 2 つあります。MKLGetVersionString 関数を使用してバージョン文字列を取得する方法と、MKLGetVersion 関数を使用してバージョン情報を含む MKLVersion 構造を取得する方法です。関数の説明と呼び出し構文は、*『インテル® MKL リファ*

*レンス・マニュアル』の「サポート関数」*を参照してください。バージョン情報を取得するプログラムサンプルが、`examples/versionquery` ディレクトリーに含まれています。このディレクトリーには、サンプルを自動的にビルドして現在のライブラリーのバージョン情報を含む要約ファイルを出力する makefile も提供されています。

サポートするコンパイラー

インテル ® MKL は、*リリースノート*に記述されているコンパイラーのみをサポートしています。しかし、ライブラリーはほかのコンパイラーでも動作することが確認されています。

CBLAS インターフェイスを使用している場合、ヘッダーファイル `mkl.h` により、すべての関数の列挙値とプロトタイプが指定されるため、プログラム開発が単純化されます。ヘッダーはプログラムが C++ コンパイラーでコンパイルされているかどうかを判断し、コンパイルされている場合、インクルード・ファイルは C++ コンパイル用に設定されます。

インテル ® MKL 9.1 以降、インテル ® Fortran 9.1 コンパイラーと関数の呼び出し規則が異なる GNU gfortran* コンパイラーを完全にサポートしています。Absoft Fortran コンパイラーも同様にサポートしています。

インテル ® MKL の使用を開始する前に

インテル ® MKL の使用を開始する前に、いくつかの重要な基本的な概念に目を通すようにしてください。

次の表は、インテル ® MKL の使用を開始する前に知っておく必要がある重要な項目を要約したものです。

表 2-1　　開始前に知っておく必要がある項目

ターゲット・プラットフォーム	ターゲットマシンのアーキテクチャーを特定します。 • IA-32 • インテル ® 64 • IA-64 (Itanium® プロセッサー・ファミリー) 理由。アプリケーションとインテル ® MKL ライブラリーをリンクする際に、使用するアーキテクチャーに対応するディレクトリーをリンクコマンドで指定する必要があるためです (「リンクするライブラリーの選択」を参照)。

表 2-1 開始前に知っておく必要がある項目 (続き)

数学問題	解いている問題で必要なインテル ® MKL 関数領域をすべて特定します。 • BLAS • スパース BLAS • LAPACK • ScaLAPACK • スパース・ソルバー・ルーチン • ベクトル数学ライブラリー関数 • ベクトル・スタティスティカル・ライブラリー関数 • フーリエ変換関数 (FFT) • クラスター FFT • 区間ソルバールーチン • 三角変換ルーチン • ポアソン、ラプラス、およびヘルムホルツ・ソルバー・ルーチン • 最適化 (Trust-Region) ソルバールーチン 理由。使用する関数領域を特定することで、*『インテル ® MKL リファレンス・マニュアル』*でルーチンを検索する項目が少なくなります。また、アプリケーションとインテル ® MKL ソフトウェアのリンクに使用するリンク行は使用する関数領域に依存します (「インテル ® マス・カーネル・ライブラリー・クラスター・ソフトウェアの使用」を参照)。コーディングのヒントは関数領域に依存することにも注意してください (「パフォーマンスを向上するためのヒントと手法」を参照)。
プログラミング言語	インテル ® MKL は Fortran と C/C++ プログラミングの両方をサポートしますが、すべての関数領域が特定の言語環境 (例えば、C/C++ または Fortran90/95) をサポートするとは限りません。使用する関数領域でサポートされる言語インターフェイスを特定してください (「インテル ® マス・カーネル・ライブラリー言語インターフェイスのサポート」を参照)。 理由。関数領域が必要な環境を直接サポートしていない場合、混在言語プログラミングを使用できます。「混在言語プログラミングとインテル ® MKL」を参照してください。 言語固有のライブラリー・インターフェイスとモジュールの一覧、および使用例は、「言語固有インターフェイスとインテル ® MKL の使用」を参照してください。
スレッド化モデル	アプリケーションをスレッド化するかどうかを以下のオプションから選択します。 • アプリケーションはすでにスレッド化されている • レガシー OpenMP* ランタイム・ライブラリー (`libguide`)、互換 OpenMP ランタイム・ライブラリー (`libiomp`)、またはサードパーティーのコンパイラーで提供されているスレッド化機能を使用してアプリケーションをスレッド化する • アプリケーションをスレッド化しない 理由。OpenMP は、インテル ® MKL が使用するスレッドの数を自動的に設定します。異なる数が必要な場合、プログラマーが利用可能な方法を使用して数を設定する必要があります。詳細 (特に、スレッド化された環境での競合を回避する方法) は、「インテル ® MKL 並列処理の使用」を参照してください。さらに、アプリケーションのスレッド化に使用するコンパイラーは、アプリケーションとリンクするスレッド化ライブラリーを決定します (「スレッド化ライブラリーのリンク」を参照)。
リンクモデル	アプリケーションとインテル ® MKL ライブラリーをリンクする適切なリンクモデルを決定します。 • スタティック • ダイナミック 理由。各リンクモデルの利点、リンクコマンドの構文と例、その他のリンクに関する情報 (カスタム・ダイナミック・ライブラリーを作成してディスク容量を節約する方法など) は、「アプリケーションと インテル ® マス・カーネル・ライブラリーのリンク」を参照してください。

表 2-1 開始前に知っておく必要がある項目 (続き)

使用する MPI	理由。アプリケーションと ScaLAPACK やクラスター FFT をリンクする際に、使用する MPI に対応するライブラリーをリンク行で指定する必要があるためです (「インテル ® マス・カーネル・ライブラリー・クラスター・ソフトウェアの使用」を参照)。

インテル ® マス・カーネル・ライブラリーの構造

3

本章は、インテル ® マス・カーネル・ライブラリー (インテル ® MKL) の構造、特にインストール後のインテル ® MKL ディレクトリーの構造と、ライブラリーのバージョンおよび種類について説明します。

インテル ® MKL は、バージョン 10.0 からレイヤーモデル (詳細は「レイヤーモデルの概念」を参照) を採用しています。この大幅な設計の変更により、ライブラリーの構造が合理化され、サイズが減少し、使用法に柔軟性が増しました。

高レベル・ディレクトリー構造

表 3-1 は、インストール後のインテル ® MKL の高レベル・ディレクトリー構造を示しています。

表 3-1 高レベル・ディレクトリー構造

ディレクトリー	内容
<*mkl ディレクトリー*>	メイン・ディレクトリー。デフォルトの場合、"/opt/intel/mkl/10.0.xxx" (xxx はインテル ® MKL パッケージ番号) 例 : "/opt/intel/mkl/10.0.039"
<*mkl ディレクトリー*>/doc	ドキュメント・ディレクトリー
<*mkl ディレクトリー*>/man/man3	インテル ® MKL 関数の man ページ
<*mkl ディレクトリー*>/examples	サンプルのソースとデータ
<*mkl ディレクトリー*>/include	ライブラリー・ルーチンとサンプルプログラムのテスト用のインクルード・ファイル
<*mkl ディレクトリー*>/interfaces/blas95	BLAS 用 Fortran 90 ラッパーとライブラリー・ビルド用のメイクファイル
<*mkl ディレクトリー*>/interfaces/lapack95	LAPACK 用 Fortran 90 ラッパーとライブラリー・ビルド用のメイクファイル
<*mkl ディレクトリー*>/interfaces/fftw2xc	インテル ® MKL FFT を呼び出す FFTW バージョン 2.x 用のラッパー (C インターフェイス)
<*mkl ディレクトリー*>/interfaces/fftw2xf	インテル ® MKL FFT を呼び出す FFTW バージョン 2.x 用のラッパー (Fortran インターフェイス)
<*mkl ディレクトリー*>/interfaces/fftw3xc	インテル ® MKL FFT を呼び出す FFTW バージョン 3.x 用のラッパー (C インターフェイス)

表 3-1 高レベル・ディレクトリー構造 (続き)

ディレクトリー	内容
<*mkl ディレクトリー*>/interfaces/fftw3xf	インテル ® MKL FFT を呼び出す FFTW バージョン 3.x 用のラッパー (Fortran インターフェイス)
<*mkl ディレクトリー*>/interfaces/fftw2x_cdft	インテル ® MKL クラスター FFT を呼び出す MPI FFTW バージョン 2.x 用のラッパー
<*mkl ディレクトリー*>/tests	テスト用のソースおよびデータ
<*mkl ディレクトリー*>/lib/32	IA-32 アーキテクチャー用のスタティック・ライブラリーと共有オブジェクト
<*mkl ディレクトリー*>/lib/em64t	インテル ® 64 アーキテクチャー (旧称インテル ® EM64T) 用のスタティック・ライブラリーと共有オブジェクト
<*mkl ディレクトリー*>/lib/64	IA-64 アーキテクチャー (インテル ® Itanium® プロセッサー・ファミリー) 用のスタティック・ライブラリーと共有オブジェクト
<*mkl ディレクトリー*>/benchmarks/linpack	LINPACK ベンチマークの OMP バージョン
<*mkl ディレクトリー*>/benchmarks/mp_linpack	LINPACK ベンチマークの MPI バージョン
<*mkl ディレクトリー*>/tools/builder	カスタム・ダイナミック・リンク・ライブラリー作成用のツール
<*mkl ディレクトリー*>/tools/environment	ユーザーシェルで環境変数を設定するシェルスクリプト
<*mkl ディレクトリー*>/tools/support	インテル ® プレミアサポートにパッケージ ID とライセンスキー情報を報告するユーティリティー
<*mkl ディレクトリー*>/tools/plugins/com.intel.mkl.help	WebHelp 形式のインテル ® MKL リファレンス・マニュアルと Eclipse プラグイン (コメントは Doc_Index.htm を参照)

レイヤーモデルの概念

インテル ® マス・カーネル・ライブラリーでは、32 ビット Windows* バージョンを除き、長い間ユーザーからはその構造がわからないようになっていました。2 つのインターフェイス・ライブラリーが提供され、ユーザーは使用するライブラリーをランタイムに選択する必要がありました。両方のライブラリーとも、比較的小さく、特定の IA-32 アーキテクチャー・ベースのプロセッサーに依存しないものです。これらのファイルを使用することにより、インターフェイスに依存しない、多くのライブラリーの複製を防ぐことができるため、ライブラリーのサイズを大幅に増加させることなく、2 つの異なるコンパイラー・インターフェイス規格をサポートすることができます。

バージョン 10.0 より、インテル ® MKL は、特にコンパイラーとスレッド化でさまざまな状況をサポートするため、このアプローチを拡張しています。

インターフェイス： IA-64 アーキテクチャー・ベースの Linux* システムでは、インテル ® Fortran コンパイラーは gnu およびその他の特定のコンパイラーと異なる複素数値を返します。これらの違いに対応するために、ライブラリーを複製するのではなく、個別のインターフェイス・ライブラリーを提供します。このため、ライブラリーのサイズを増加させることなく、コンパイラー間

の相違に対応します。同様に、LP64 はインターフェイスを利用して ILP64 の上でサポートされます。また、単精度が 64 ビット演算を意味する古いスーパーコンピューターのサポートの要望にも、インターフェイス・ライブラリーで必要なマッピングを提供することで対応しています。

スレッド化： インテル ® MKL では効率性を優先させるため、ループレベルのスレッド化ではなく、ライブラリー全体の関数レベルのスレッド化をこれまで使用してきました。従って、スレッド化はすべて比較的小さな関数群に集約でき、ライブラリーにまとめることができます。コンパイラー固有のランタイム・ライブラリーへの参照はすべて、これらの関数で生成されます。これらの関数を異なるコンパイラーでコンパイルして、サポートしているそれぞれのコンパイラーに対してスレッド化ライブラリー・レイヤーを提供することにより、インテル ® MKL は、インテル ® コンパイラー以外のコンパイラーを使用してスレッド化されたプログラムで動作します。スレッド化はすべて OpenMP* により提供されますが、スレッド化をオフにしてこのレイヤーをコンパイルすると、スレッド化されていないレイヤーを通じてライブラリーの非スレッド化バージョンも提供できます。

計算： 指定のプロセッサー・ファミリー (IA-32、IA-63、またはインテル ® 64 アーキテクチャー・ベースのプロセッサー) で、単一の計算ライブラリーがすべてのインターフェイスとスレッド化レイヤーで使用されます。計算レイヤーでの並列処理は行われません。

ランタイム・ライブラリー (RTL): 最後のレイヤーは RTL サポートを提供します。すべての RTL がインテル ® MKL で提供されるとは限りません。インテル ® MKL クラスター・ソフトウェアに関連するものを除いて、インテル ® コンパイラー・ベースの RTL であるインテル ® レガシー OpenMP ランタイム・コンパイラー・ライブラリー (`libguide`) とインテル ® 互換 OpenMP ランタイム・コンパイラー・ライブラリー (`libiomp`) のみ提供されます。インテル ® コンパイラー以外のスレッド化コンパイラーを使用してスレッド化を行うには、スレッド化レイヤー・ライブラリーを利用するか、適切な状況で互換ライブラリーを利用します。

レイヤー

ライブラリーには 4 つのレイヤーがあります。

1. インターフェイス・レイヤー
2. スレッド化レイヤー
3. 計算レイヤー
4. コンパイラー・サポート RTL レイヤー (RTL レイヤー)

インターフェイス・レイヤー： このレイヤーは、コンパイルされたアプリケーションのコードとライブラリーのスレッド化および計算部分を本質的に一致させます。このレイヤーは、一致のために以下の項目を提供します。

- インテル ® MKL ILP64 ソフトウェアへの LP64 インターフェイス
  (詳細は「ILP64 プログラミングのサポート」を参照)
- 異なるコンパイラーが関数値を返す方法に対処する手段
- Cray 形式の名前を使用するソフトウェア・ベンダー向けに、ILP64 を使用するアプリケーションの倍精度名のマッピングに対する単精度名のマッピング

スレッド化レイヤー： このレイヤーは、スレッド化されたインテル® MKL をサポートしているスレッド化コンパイラーで共有する方法を提供します。また、ライブラリーの逐次バージョンも提供します。従来はライブラリーに対して内部的に行われていたことが、今では異なる環境 (スレッド化または逐次) やコンパイラー (インテル、gnu、その他) 用にコンパイルすることにより、スレッド化レイヤーで参照することができます。

計算レイヤー： インテル® MKL の中心であり、32 ビットのインテル® プロセッサーと 32 ビット・オペレーティング・システムのように、任意のプロセッサー / オペレーティング・システム・ファミリーで 1 つの variant のみを持ちます。計算レイヤーは、アーキテクチャーまたはアーキテクチャーの機能を識別することで、実行時にさまざまなアーキテクチャー用に適切なバイナリーコードを選択します。インテル® MKL は、異なる計算環境に影響されない、大規模な計算レイヤーと見なすことができます。計算レイヤーには RTL 要件がないため、RTL は、計算レイヤーの上のレイヤー (インターフェイス・レイヤーまたはスレッド化レイヤー) の 1 つを指します。ほとんどの場合、RTL レイヤーはスレッド化レイヤーと一致します。

コンパイラー・サポート RTL レイヤー： このレイヤーには、ランタイム・ライブラリー・サポート関数が含まれます。例えば、`libguide` や `libiomp` は、インテル® MKL で OpenMP のスレッド化サポートを提供する RTL です。

第 5 章の「スレッド化ライブラリーのリンク」セクションも参照してください。

ライブラリーの逐次バージョン

バージョン 9.1 から、インテル® MKL パッケージはライブラリーの逐次 (非スレッド) バージョンをサポートしています。コンパイラー・サポート RTL レイヤー (レガシー OpenMP ランタイム・ライブラリーまたは互換 OpenMP ランタイム・ライブラリー) を必要とせず、`OMP_NUM_THREADS` 環境変数には応答しません (詳細は、第 6 章の「インテル® MKL 並列処理の使用」セクションを参照)。このバージョンのインテル® MKL は、非スレッドコードを実行しますが、このコードはスレッドセーフなので、OpenMP コードの並列領域で使用できます。インテル® MKL のスレッド化を使用しない特別な理由がある場合のみ、逐次バージョンを使用してください。逐次バージョン (レイヤー) は、インテル® MKL をインテル以外のコンパイラーでスレッド化されたプログラムと使用する場合や、さまざまな理由によりライブラリーの非スレッドバージョンが必要な場合に役立ちます。詳細は、第 6 章の「実行環境における競合の回避」セクションを参照してください。

インテル® MKL の逐次バージョンを使用するには、スレッド化レイヤーで、リンクに使用するライブラリーとして `*sequential.*` ライブラリーを選択します (「詳細なディレクトリー構造」を参照)。

逐次ライブラリーは、インテル® MKL ソフトウェアをスレッドセーフにするためにリンク行で指定する、POSIX スレッド・ライブラリー (`pthread`) に依存することに注意してください (第 5 章の「その他のリンクの例」を参照)。

ILP64 プログラミングのサポート

"LP64" と "ILP64" という用語は、歴史的な背景と次の Web サイトに記述されているプログラミング・モデルの原理により使用されています。
http://www.unix.org/version2/whatsnew/lp64_wp.html (英語)

インテル ® MKL ILP64 ライブラリーは、プログラミング・モデルの原理に完全に沿っているわけではありませんが、一般的な考えは同じです。大規模な配列 (2^{31}-1 以上の要素を含む配列) のインデックス処理には 64 ビット整数型を使用します。

LP64 インターフェイスと ILP64 インターフェイスは、インターフェイス・レイヤーでサポートされています。インターフェイス・レイヤーで適切なライブラリーが選択されると (「詳細なディレクトリー構造」を参照)、インターフェイス・レイヤーの下にあるすべてのライブラリーは、選択したインターフェイスを使用してコンパイルされます。

ILP64 インターフェイスと LP64 インターフェイスの違いは *『インテル ® MKL リファレンス・マニュアル』* で説明されていないため、ILP64 インターフェイスの詳細は、以下のディレクトリーにあるインクルード・ファイル、サンプル、およびテストを参照してください。

<mkl ディレクトリー>/include

<mkl ディレクトリー>/examples

<mkl ディレクトリー>/tests

このセクションでは、以下の内容を説明します。

- インテル ® MKL における ILP64 の概念の実装
- ILP64 インターフェイス用にコードをコンパイルする方法
- ILP64 インターフェイス用にコードを記述する方法
- ILP64 インターフェイス用のインテル ® MKL インクルード・ファイルを参照する方法

また、ILP64 サポートの制限についても説明します。

概念

ILP64 インターフェイスは、以下の 2 つの理由により提供されています。

- 大規模なデータ配列 (要素数 20 億以上) をサポートする
- `-i8` コンパイラー・オプションを使用して Fortran コードをコンパイルできるようにする

インテル ® Fortran コンパイラーは `INTEGER` 型の動作を変更するために `-i8` オプションをサポートしています。デフォルトでは、標準 `INTEGER` 型は 4 バイトです。`-i8` オプションを使用すると、コンパイラーは `INTEGER` 定数、変数、関数およびサブルーチン・パラメーターを 8 バイトとして処理します。

ILP64 バイナリー・インターフェイスは、配列サイズ、インデックス、ストライドなどを定義する関数パラメーターに 8 バイトの整数を使用します。言語レベル、つまり、インテル ® MKL のインクルード・ディレクトリーに含まれている `*.f90` ファイルや `*.fi` ファイルでは、これらのパラメーターは `INTEGER` として宣言されます。

Fortran コードを IPL64 インターフェイスで使用するには、`-i8` コンパイラー・オプションを使用してコードをコンパイルする必要があります。逆に、`-i8` オプションを使用してコードをコンパイルした場合、LP64 バイナリー・インターフェイスでは `INTEGER` 型は 4 バイトでなければならないため、ILP64 インターフェイスでしか使用できません。

一部のインテル ® MKL 関数とサブルーチンは、コードが `-i8` オプションを使用してコンパイルされたかどうかに関係なく、常に 4 バイトの `INTEGER*4` または `INTEGER(KIND=4)` 型のスカラー / 配列パラメーターを使用することに注意してください。

C 言語および C++ 言語の場合、インテル® MKL は Fortran の INTEGER 型に相当する MKL_INT 型を提供します。MKL_INT マクロは、デフォルトでは標準 C/C++ の int 型として定義されますが、コードのコンパイルで MKL_ILP64 マクロが定義されている場合、MKL_INT は 64 ビット整数型として定義されます。MKL_ILP64 マクロを定義するには、-DMKL_ILP64 コマンドライン・オプションを使用してコンパイラーを呼び出します。

インテル® MKL は、C/C++ 用の特定の FFT インターフェイスで ILP64 インターフェイスを維持するために、MKL_LONG 型も定義します。MKL_LONG マクロは、デフォルトでは標準 C/C++ の long 型として定義されますが、コードのコンパイルで MKL_ILP64 マクロが定義されている場合、MKL_LONG は 64 ビット整数型として定義されます。

注 : int 型は、最近のほとんどの C/C++ コンパイラーと同様に 32 ビットです。long 型は、特定の OS に依存し、32 ビットまたは 64 ビットです。

C/C++ 言語用のインテル® MKL インターフェイス (インテル® MKL のインクルード･ディレクトリーにある *.h ヘッダーファイル) では、配列サイズ、インデックス、ストライドなどの関数パラメーターは MKL_INT として宣言されます。

C/C++ 用の FFT インターフェイスは特別なケースで、mkl_dfti.h ヘッダーファイル で、MKL_LONG 型をインターフェイス関数の明示的パラメーターと暗黙的パラメーターの両方で使用します。特に、DftiCreateDescriptor() 関数の明示的パラメーター *dimension* の型は MKL_LONG で、暗黙的パラメーター *length* の型は 1 次元変換の場合は MKL_LONG、複数次元変換の場合は MKL_LONG[] (MKL_LONG 型の数の配列) です。

C/C++ コードを ILP64 インターフェイスで使用するには、-DMKL_ILP64 コマンドライン・オプションを使用して MKL_INT および MKL_LONG を 64 ビットにする必要があります。逆に、-DMKL_ILP64 オプションを使用してコードをコンパイルした場合、LP64 バイナリー・インターフェイスでは MKL_INT は 32 ビット、MKL_LONG は標準 long 型でなければならないため、ILP64 インターフェイスでしか使用できません。

特定の MKL 関数では、パラメーターは int または int[] として明示的に宣言されることに注意してください。これらの整数パラメーターは、-DMKL_ILP64 オプションを使用してコードをコンパイルしたかどうかに関係なく、常に 32 ビットです。

表 3-2 は、インテル® MKL で ILP64 の概念がどのように実装されているかを要約したものです。

表 3-2　　インテル® MKL の ILP64 の概念

	Fortran	C/C++
ILP64 インターフェイスと LP64 インターフェイスで同じインクルード・ディレクトリー	*<mkl ディレクトリー>*/include	
常に 32 ビットのパラメーターに使用される型	INTEGER*4	int
64 ビット整数 (ILP64 インターフェイス) および 32 ビット整数 (LP64 インターフェイス) のパラメーターに使用される型	INTEGER	MKL_INT

表 3-2 インテル ® MKL の ILP64 の概念 (続き)

	Fortran	C/C++
FFT 関数のすべての整数パラメーターに使用される型	INTEGER	MKL_LONG
ILP64 のコンパイルを制御するコマンドライン・オプション	-i8	-DMKL_ILP64

ILP64 用のコンパイル

インテル ® MKL インクルード・ディレクトリーの同じコピーが ILP64 インターフェイスと LP64 インターフェイスの両方で使用されます。ILP64 インターフェイス用のコンパイルは次のようになります。

Fortran:

```
ifort -i8 -I<mkl ディレクトリー>/include …
```

C/C++:

```
icc -DMKL_ILP64 -I<mkl ディレクトリー>/include …
```

LP64 インターフェイス用にコンパイルするには、-i8 または -DMKL_ILP64 オプションを省略してください。
-i8 または -DMKL_ILP64 オプションを使用してコンパイルしたアプリケーションと LP64 ライブラリーをリンクすると、予測不可能な結果や誤出力が発生する場合があります。

表 3-3 は、コンパイラー・オプションを要約したものです。

表 3-3 ILP64 インターフェイス用のコンパイラー・オプション

	Fortran	C/C++
ILP64 インターフェイス	ifort -i8 ...	icc -DMKL_ILP64 ...
LP64 インターフェイス	ifort ...	icc ...

ILP64 用のコーディング

インテル ® MKL インクルード・ディレクトリーの *.f90、*.fi、および *.h ファイルは ILP64 インターフェイスの要件を満たすように変更されていますが、LP64 インターフェイスは変更されていません。つまり、32 ビット整数だった関数パラメーターはすべて 32 ビット整数型のままで、標準 long 整数だった関数パラメーターはすべて標準 long 型のままです。このため、ILP64 インターフェイスを使用していなければ、既存コードの各行を変更する必要はありません。

ILP64 へ変更したり ILP64 用に新しいコードを記述する場合は、インテル ® MKL 関数とサブルーチンのパラメーターに適切な型を使用する必要があります。ILP64 で 64 ビット整数でなければならないパラメーターについては、以下の汎用整数型を使用してください。

- INTEGER (Fortran の場合)
- MKL_INT (C/C++ の場合)
- MKL_LONG (C/C++ FFT インターフェイスのパラメーターの場合)

この方法で記述されたコードは、ILP64 と LP64 インターフェイスの両方で動作します。

ILP64 で 64 ビットでなければならない整数パラメーターにほかの 64 ビットの型を使用することもできます。例えば、インテル ® コンパイラーでは、以下の型を使用できます。

- `INTEGER(KIND=8)` (Fortran の場合)
- `long long int` (C/C++ の場合)

この方法で記述されたコードは、LP64 インターフェイスでは動作しないことに注意してください。表 3-4 は、整数型の使用法を要約したものです。

表 3-4 整数型

	Fortran	C/C++

32 ビット整数	`INTEGER*4`
	または
	`INTEGER(KIND=4)`
		`int`

汎用整数 : ▪ 64 ビット (ILP64 の場合) ▪ 32 ビット (その他の場合)	`INTEGER` `KIND` の指定なし	`MKL_INT`
FFT インターフェイス・パラメーター用のユニバーサル型	`INTEGER` `KIND` の指定なし	`MKL_LONG`

インテル ® MKL インクルード・ファイルの参照

整数パラメーターを使用する関数について、*『インテル ® MKL リファレンス・マニュアル』*では、ILP64 で 64 ビットになるパラメーターと 32 ビットのままのパラメーターは説明されていません。

この情報については、インクルード・ファイル、サンプルおよびテストを参照する必要があります。まず最初に、インクルード・ファイルを参照してください。インクルード・ファイルには、すべてのインテル ® MKL 関数のプロトタイプが含まれています。次に、関数の使用法を理解するためにサンプルとテストを参照してください。

インクルード・ファイルはすべて *<mkl ディレクトリー>*`/include` ディレクトリーにあります。表 3-5 は、参照するインクルード・ファイルを示しています。

表 3-5 インテル ® MKL インクルード・ファイル

関数領域	インクルード・ファイル	
	Fortran	C/C++
BLAS ルーチン	`mkl_blas.f90` `mkl_blas.fi`	`mkl_blas.h`
BLAS に対する CBLAS インターフェイス		`mkl_cblas.h`
スパース BLAS ルーチン	`mkl_spblas.fi`	`mkl_spblas.h`
LAPACK ルーチン	`mkl_lapack.f90` `mkl_lapack.fi`	`mkl_lapack.h`

表 3-5 インテル ® MKL インクルード・ファイル (続き)

関数領域	インクルード・ファイル	
	Fortran	C/C++
ScaLAPACK ルーチン		mkl_scalapack.h
スパース・ソルバー・ルーチン		
▪ PARDISO	mkl_pardiso.f77 mkl_pardiso.f90	mkl_pardiso.h
▪ DSS インターフェイス	mkl_dss.f77 mkl_dss.f90	mkl_dss.h
▪ RCI 反復ソルバー ▪ ILU 因数分解	mkl_rci.fi	mkl_rci.h
最適化ソルバールーチン	mkl_rci.fi	mkl_rci.h
ベクトル数学関数	mkl_vml.fi	mkl_vml_functions.h
ベクトル・スタティスティカル関数	mkl_vsl.fi mkl_vsl_subroutine.fi	mkl_vsl_functions.h
フーリエ変換関数	mkl_dfti.f90	mkl_dfti.h
クラスターフーリエ変換関数	mkl_cdft.f90	mkl_cdfti.h
偏微分方程式サポートルーチン		
▪ 三角変換	mkl_trig_transforms.f90	mkl_trig_transforms.h
▪ ポアソンソルバー	mkl_poisson.f90	mkl_poisson.h

Fortran インターフェイスのみをサポートする一部の関数領域 (表 A-1 を参照) でも、インクルード・ディレクトリーに C/C++ 用のヘッダーファイルが提供されます。この *.h ファイルを使用すると、C/C++ コードから Fortran バイナリー・インターフェイスが利用可能になるため、ILP64 を含む C インターフェイスを記述できます。

制限

すべてのコンポーネントが ILP64 機能をサポートしているとは限らないことに注意してください。表 3-6 は、ILP64 インターフェイスをサポートしている関数領域を示しています。

表 3-6 インテル ® MKL の ILP64 サポート

関数領域	ILP64 のサポート
BLAS	○
スパース BLAS	○
LAPACK	○
ScaLAPACK	○
VML	○
VSL	○
PARDISO ソルバー	○

表 3-6 インテル ® MKL の ILP64 サポート (続き)

関数領域	ILP64 のサポート
DSS ソルバー	○
ISS ソルバー	○
最適化 (Trust-Region) ソルバー	○
FFT	○
FFTW	X
クラスター FFT	○
PDE サポート : 三角変換	○
PDE サポート : ポアソンソルバー	○
GMP	X
区間演算	X
BLAS 95	○
LAPACK 95	○

インテル ® MKL のバージョン

インテル ® MKL Linux 版は、以下のバージョンを識別します。

- IA-32 アーキテクチャーの場合、バージョンは `lib/32` ディレクトリーに含まれています。
- インテル ® 64 アーキテクチャーの場合、バージョンは `lib/em64t` ディレクトリーに含まれています。
- IA-64 アーキテクチャーの場合、バージョンは `lib/64` ディレクトリーに含まれています。

これらのディレクトリーの詳細な構造は、表 3-7 を参照してください。

詳細なディレクトリー構造

以下の表の情報は、ライブラリーのアーキテクチャー固有ディレクトリーの詳細な構造を示しています。`doc` ディレクトリーの内容は、「ドキュメント・ディレクトリーの内容」セクションを参照してください。`benchmarks` ディレクトリーのサブディレクトリーの内容は、第 10 章を参照してください。

表 3-7 詳細なディレクトリー構造

ディレクトリー / ファイル	内容
`lib/32`[1]	IA-32 アーキテクチャー用のすべてのライブラリー
スタティック・ライブラリー	
インターフェイス・レイヤー	
`libmkl_intel.a`	インテル ® コンパイラー用インターフェイス・ライブラリー
`libmkl_gf.a`	GNU Fortran コンパイラー用インターフェイス・ライブラリー

表 3-7 詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
スレッド化レイヤー	
libmkl_intel_thread.a	インテル ® コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_gnu_thread.a	GNU コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_sequential.a	逐次ドライバー・ライブラリー
計算レイヤー	
libmkl_core.a	IA-32 アーキテクチャー用カーネル・ライブラリー
libmkl_ia32.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_lapack.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_solver.a	スパースソルバー、区間ソルバー、および GMP ルーチン
libmkl_solver_sequential.a	スパースソルバー、区間ソルバー、および GMP ルーチン・ライブラリーの逐次バージョン
libmkl_scalapack.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_scalapack_core.a	ScaLAPACK ルーチン
libmkl_cdft.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_cdft_core.a	FFT のクラスターバージョン
RTL レイヤー	
libguide.a	スタティック・リンク用インテル ® レガシー OpenMP ランタイム・ライブラリー
libiomp5.a	スタティック・リンク用インテル ® 互換 OpenMP ランタイム・ライブラリー
libmkl_blacs.a	以下の MPICH バージョンをサポートする BLACS ルーチン。 ▪ Topspin* MPICH バージョン 1.2.5 (*ch_vapi* デバイスで構成) ▪ Myricom* MPICH バージョン 1.2.5.10 ▪ ANL* MPICH バージョン 1.2.5.2
libmkl_blacs_intelmpi.a	インテル ® MPI 1.0 をサポートする BLACS ルーチン
libmkl_blacs_intelmpi20.a	インテル ® MPI 2.0/3.0、および MPICH 2.0 をサポートする BLACS ルーチン
libmkl_blacs_openmpi.a	OpenMPI をサポートする BLACS ルーチン
ダイナミック・ライブラリー	
インターフェイス・レイヤー	
libmkl_intel.so	インテル ® コンパイラー用インターフェイス・ライブラリー

表 3-7 詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
libmkl_gf.so	GNU Fortran コンパイラー用インターフェイス・ライブラリー
スレッド化レイヤー	
libmkl_intel_thread.so	インテル ® コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_gnu_thread.so	GNU コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_sequential.so	逐次ドライバー・ライブラリー
計算レイヤー	
libmkl.so	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_core.so	プロセッサー固有カーネル・ライブラリーのダイナミック・ロード用ライブラリー・ディスパッチャー
libmkl_def.so	デフォルト・カーネル・ライブラリー (インテル ® Pentium® プロセッサー、インテル ® Pentium® Pro プロセッサー、インテル ® Pentium® II プロセッサー)
libmkl_p3.so	インテル ® Pentium® III プロセッサー用カーネル・ライブラリー
libmkl_p4.so	インテル ® Pentium® 4 プロセッサー用カーネル・ライブラリー
libmkl_p4p.so	ストリーミング SIMD 拡張命令 3 (SSE3) 対応インテル ® Pentium® 4 プロセッサー用カーネル・ライブラリー
libmkl_p4m.so	インテル ® Core™ マイクロアーキテクチャー・ベースのプロセッサー用カーネル・ライブラリー (mkl_p4p.so が対応しているインテル ® Core™ Duo プロセッサーおよびインテル ® Core™ Solo プロセッサーを除く)
libmkl_lapack.so	LAPACK ルーチンとドライバー
libmkl_ias.so	区間演算ルーチン
libmkl_vml_def.so	古いインテル ® Pentium® プロセッサー用デフォルトカーネルの VML/VSL 部分
libmkl_vml_ia.so	新しいインテル ® アーキテクチャー・プロセッサー用 VML/VSL デフォルトカーネル
libmkl_vml_p3.so	インテル ® Pentium® III プロセッサー用カーネルの VML/VSL 部分
libmkl_vml_p4.so	インテル ® Pentium® 4 プロセッサー用カーネルの VML/VSL 部分
libmkl_vml_p4p.so	ストリーミング SIMD 拡張命令 3 (SSE3) 対応インテル ® Pentium® 4 プロセッサー用 VML/VSL
libmkl_vml_p4m.so	インテル ® Core™ マイクロアーキテクチャー・ベースのプロセッサー用 VML/VSL
libmkl_vml_p4m2.so	45nm Hi-k インテル ® Core™2 プロセッサー・ファミリーおよびインテル ® Xeon® プロセッサー・ファミリー用 VML/VSL
RTL レイヤー	
libguide.so	ダイナミック・リンク用インテル ® レガシー OpenMP ランタイム・ライブラリー

表 3-7 詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
libiomp5.so	ダイナミック・リンク用インテル ® 互換 OpenMP ランタイム・ライブラリー
lib/em64t[1]	インテル ® 64 アーキテクチャー用のすべてのライブラリー
スタティック・ライブラリー	
インターフェイス・レイヤー	
libmkl_intel_ilp64.a	インテル ® コンパイラー用 ILP64 インターフェイス・ライブラリー
libmkl_intel_lp64.a	インテル ® コンパイラー用 LP64 インターフェイス・ライブラリー
libmkl_intel_sp2dp.a	インテル ® コンパイラー用 SP2DP インターフェイス・ライブラリー
libmkl_gf_ilp64.a	GNU Fortran コンパイラー用 ILP64 インターフェイス・ライブラリー
libmkl_gf_lp64.a	GNU Fortran コンパイラー用 LP64 インターフェイス・ライブラリー
スレッド化レイヤー	
libmkl_intel_thread.a	インテル ® コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_gnu_thread.a	GNU コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_sequential.a	逐次ドライバー・ライブラリー
計算レイヤー	
libmkl_core.a	インテル ® 64 アーキテクチャー用カーネル・ライブラリー
libmkl_em64t.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_lapack.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_solver.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_solver_lp64.a	LP64 インターフェイスをサポートするスパースソルバー、区間ソルバー、および GMP ルーチン・ライブラリー
libmkl_solver_ilp64.a	ILP64 インターフェイスをサポートするスパース・ソルバー・ルーチン・ライブラリー
libmkl_solver_lp64_sequential.a	LP64 インターフェイスをサポートするスパースソルバー、区間ソルバー、および GMP ルーチン・ライブラリーの逐次バージョン
libmkl_solver_ilp64_sequential.a	ILP64 インターフェイスをサポートするスパース・ソルバー・ルーチン・ライブラリーの逐次バージョン
libmkl_scalapack.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_scalapack_lp64.a	LP64 インターフェイスをサポートする ScaLAPACK ルーチン・ライブラリー

表 3-7　　詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
libmkl_scalapack_ilp64.a	ILP64 インターフェイスをサポートする ScaLAPACK ルーチン・ライブラリー
libmkl_cdft.a	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_cdft_core.a	FFT のクラスターバージョン
RTL レイヤー	
libguide.a	スタティック・リンク用インテル ® レガシー OpenMP ランタイム・ライブラリー
libiomp5.a	スタティック・リンク用インテル ® 互換 OpenMP ランタイム・ライブラリー
libmkl_blacs_ilp64.a	以下の MPICH バージョンをサポートする BLACS ルーチンの ILP64 バージョン。 ▪ Topspin* MPICH バージョン 1.2.5 (*ch_vapi* デバイスで構成) ▪ Myricom* MPICH バージョン 1.2.5.10 ▪ ANL* MPICH バージョン 1.2.5.2
libmkl_blacs_lp64.a	以下の MPICH バージョンをサポートする BLACS ルーチンの LP64 バージョン。 ▪ Topspin* MPICH バージョン 1.2.5 (*ch_vapi* デバイスで構成) ▪ Myricom* MPICH バージョン 1.2.5.10 ▪ ANL* MPICH バージョン 1.2.5.2
libmkl_blacs_intelmpi_ilp64.a	インテル ® MPI 1.0 をサポートする BLACS ルーチンの ILP64 バージョン
libmkl_blacs_intelmpi_lp64.a	インテル ® MPI 1.0 をサポートする BLACS ルーチンの LP64 バージョン
libmkl_blacs_intelmpi20_ilp64.a	インテル ® MPI 2.0/3.0、および MPICH 2.0 をサポートする BLACS ルーチンの ILP64 バージョン
libmkl_blacs_intelmpi20_lp64.a	インテル ® MPI 2.0/3.0、および MPICH 2.0 をサポートする BLACS ルーチンの LP64 バージョン
libmkl_blacs_openmpi_ilp64.a	OpenMPI をサポートする BLACS ルーチンの ILP64 バージョン
libmkl_blacs_openmpi_lp64.a	OpenMPI をサポートする BLACS ルーチンの LP64 バージョン
ダイナミック・ライブラリー	
インターフェイス・レイヤー	
libmkl_intel_ilp64.so	インテル ® コンパイラー用 ILP64 インターフェイス・ライブラリー
libmkl_intel_lp64.so	インテル ® コンパイラー用 LP64 インターフェイス・ライブラリー
libmkl_intel_sp2dp.so	インテル ® コンパイラー用 SP2DP インターフェイス・ライブラリー
libmkl_gf_ilp64.so	GNU Fortran コンパイラー用 ILP64 インターフェイス・ライブラリー

表 3-7 詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
libmkl_gf_lp64.so	GNU Fortran コンパイラー用 LP64 インターフェイス・ライブラリー
スレッド化レイヤー	
libmkl_intel_thread.so	インテル ® コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_gnu_thread.so	GNU コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_sequential.so	逐次ドライバー・ライブラリー
計算レイヤー	
libmkl.so	インテル ® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_core.so	プロセッサー固有カーネル・ライブラリーのダイナミック・ロード用ライブラリー・ディスパッチャー
libmkl_def.so	デフォルト・カーネル・ライブラリー
libmkl_p4n.so	インテル ® 64 アーキテクチャー対応インテル ® Xeon® プロセッサー用カーネル・ライブラリー
libmkl_mc.so	インテル ® Core™ マイクロアーキテクチャー・ベースのプロセッサー用カーネル・ライブラリー
libmkl_lapack.so	LAPACK ルーチンとドライバー
libmkl_ias.so	区間演算ルーチン
libmkl_vml_def.so	デフォルトカーネルの VML/VSL 部分
libmkl_vml_mc.so	インテル ® Core™ マイクロアーキテクチャー・ベースのプロセッサー用 VML/VSL
libmkl_vml_p4n.so	インテル ® 64 アーキテクチャー対応インテル ® Xeon® プロセッサー用 VML/VSL
libmkl_vml_mc2.so	45nm Hi-k インテル ® Core™2 プロセッサー・ファミリーおよびインテル ® Xeon® プロセッサー・ファミリー用 VML/VSL
RTL レイヤー	
libguide.so	ダイナミック・リンク用インテル ® レガシー OpenMP ランタイム・ライブラリー
libiomp5.so	ダイナミック・リンク用インテル ® 互換 OpenMP ランタイム・ライブラリー
lib/64[1]	IA-64 アーキテクチャー用のすべてのライブラリー
スタティック・ライブラリー	
インターフェイス・レイヤー	
libmkl_intel_ilp64.a	インテル ® コンパイラー用 ILP64 インターフェイス・ライブラリー
libmkl_intel_lp64.a	インテル ® コンパイラー用 LP64 インターフェイス・ライブラリー
libmkl_intel_sp2dp.a	インテル ® コンパイラー用 SP2DP インターフェイス・ライブラリー

表 3-7 詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
libmkl_gf_ilp64.a	GNU Fortran コンパイラー用 ILP64 インターフェイス・ライブラリー
libmkl_gf_lp64.a	GNU Fortran コンパイラー用 LP64 インターフェイス・ライブラリー
スレッド化レイヤー	
libmkl_intel_thread.a	インテル® コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_gnu_thread.a	GNU コンパイラーをサポートする並列ドライバー・ライブラリー
libmkl_sequential.a	逐次ドライバー・ライブラリー
計算レイヤー	
libmkl_core.a	IA-64 アーキテクチャー用カーネル・ライブラリー
libmkl_ipf.a	インテル® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_lapack.a	インテル® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_solver.a	インテル® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_solver_lp64.a	LP64 インターフェイスをサポートするスパースソルバー、区間ソルバー、および GMP ルーチン・ライブラリー
libmkl_solver_ilp64.a	ILP64 インターフェイスをサポートするスパース・ソルバー・ルーチン・ライブラリー
libmkl_solver_lp64_sequential.a	LP64 インターフェイスをサポートするスパースソルバー、区間ソルバー、および GMP ルーチン・ライブラリーの逐次バージョン
libmkl_solver_ilp64_sequential.a	ILP64 インターフェイスをサポートするスパース・ソルバー・ルーチン・ライブラリーの逐次バージョン
libmkl_scalapack.a	インテル® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_scalapack_lp64.a	LP64 インターフェイスをサポートする ScaLAPACK ルーチン・ライブラリー
libmkl_scalapack_ilp64.a	ILP64 インターフェイスをサポートする ScaLAPACK ルーチン・ライブラリー
libmkl_cdft.a	インテル® MKL ライブラリーへの参照を含むダミー・ライブラリー
libmkl_cdft_core.a	FFT のクラスターバージョン
RTL レイヤー	
libguide.a	スタティック・リンク用インテル® レガシー OpenMP ランタイム・ライブラリー
libiomp5.a	スタティック・リンク用インテル® 互換 OpenMP ランタイム・ライブラリー

表 3-7 詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
`libmkl_blacs_ilp64.a`	以下の MPICH バージョンをサポートする BLACS ルーチンの ILP64 バージョン。 ▪ Topspin* MPICH バージョン 1.2.5 (*ch_vapi* デバイスで構成) ▪ Myricom* MPICH バージョン 1.2.5.10 ▪ ANL* MPICH バージョン 1.2.5.2
`libmkl_blacs_lp64.a`	以下の MPICH バージョンをサポートする BLACS ルーチンの LP64 バージョン。 ▪ Topspin* MPICH バージョン 1.2.5 (*ch_vapi* デバイスで構成) ▪ Myricom* MPICH バージョン 1.2.5.10 ▪ ANL* MPICH バージョン 1.2.5.2
`libmkl_blacs_intelmpi_ilp64.a`	インテル ® MPI 1.0 をサポートする BLACS ルーチンの ILP64 バージョン
`libmkl_blacs_intelmpi_lp64.a`	インテル ® MPI 1.0 をサポートする BLACS ルーチンの LP64 バージョン
`libmkl_blacs_intelmpi20_ilp64.a`	インテル ® MPI 2.0/3.0、および MPICH 2.0 をサポートする BLACS ルーチンの ILP64 バージョン
`libmkl_blacs_intelmpi20_lp64.a`	インテル ® MPI 2.0/3.0、および MPICH 2.0 をサポートする BLACS ルーチンの LP64 バージョン
`libmkl_blacs_openmpi_ilp64.a`	OpenMPI をサポートする BLACS ルーチンの ILP64 バージョン
`libmkl_blacs_openmpi_lp64.a`	OpenMPI をサポートする BLACS ルーチンの LP64 バージョン
ダイナミック・ライブラリー	
インターフェイス・レイヤー	
`libmkl_intel_ilp64.so`	インテル ® コンパイラー用 ILP64 インターフェイス・ライブラリー
`libmkl_intel_lp64.so`	インテル ® コンパイラー用 LP64 インターフェイス・ライブラリー
`libmkl_intel_sp2dp.so`	インテル ® コンパイラー用 SP2DP インターフェイス・ライブラリー
`libmkl_gf_ilp64.so`	GNU Fortran コンパイラー用 ILP64 インターフェイス・ライブラリー
`libmkl_gf_lp64.so`	GNU Fortran コンパイラー用 LP64 インターフェイス・ライブラリー
スレッド化レイヤー	
`libmkl_intel_thread.so`	インテル ® コンパイラーをサポートする並列ドライバー・ライブラリー
`libmkl_gnu_thread.so`	GNU コンパイラーをサポートする並列ドライバー・ライブラリー
`libmkl_sequential.so`	逐次ドライバー・ライブラリー

表 3-7 詳細なディレクトリー構造 (続き)

ディレクトリー / ファイル	内容
計算レイヤー	
`libmkl.so`	インテル® MKL ライブラリーへの参照を含むダミー・ライブラリー
`libmkl_core.so`	プロセッサー固有カーネル・ライブラリーのダイナミック・ロード用ライブラリー・ディスパッチャー
`libmkl_i2p.so`	IA-64 アーキテクチャー用カーネル・ライブラリー
`libmkl_lapack.so`	LAPACK ルーチンとドライバー
`libmkl_ias.so`	区間演算ルーチン
`libmkl_vml_i2p.so`	IA-64 アーキテクチャー用 VML カーネル
RTL レイヤー	
`libguide.so`	ダイナミック・リンク用インテル® レガシー OpenMP ランタイム・ライブラリー
`libiomp5.so`	ダイナミック・リンク用インテル® 互換 OpenMP ランタイム・ライブラリー

1. さらに、`interfaces` ディレクトリーのメイクファイル操作の結果、多くのインターフェイス・ライブラリーが生成されます (第 7 章の「言語固有インターフェイスとインテル® MKL の使用」セクションを参照)。

ダミー・ライブラリー

基本レイヤー・ライブラリーは、適切なライブラリーの組み合わせをより柔軟に選択できますが、リンク行のライブラリー名で下位互換性がありません。ダミー・ライブラリーは、レイヤー・ライブラリーを使用しない MKL の以前のバージョンとの下位互換性のために提供されています。

ダミー・ライブラリーには、機能ではなく、1 セットのレイヤー・ライブラリーの依存関係が含まれています。リンク行でダミー・ライブラリーを指定すると、依存するレイヤー・ライブラリーの指定を省略できます。ライブラリーは自動的にリンクされます。ダミー・ライブラリーには、以下のレイヤー・ライブラリーの依存関係が含まれています (デフォルト動作)。

- インターフェイス : Intel、LP64
- スレッド化 : Intel でコンパイル
- 計算 : 計算ライブラリー

このため、上記のインターフェイスを利用してインテル® コンパイラーで提供されている OpenMP スレッド化を使用する場合、リンク行を変更しないでください。

ドキュメント・ディレクトリーの内容

表 3-8 は、インテル ® MKL インストール・ディレクトリーの doc サブディレクトリーの内容を示しています。

表 3-8　doc ディレクトリーの内容

ファイル名	内容
mklEULA.txt	インテル ® MKL の使用許諾契約書
mklSupport.txt	テクニカルサポートで使用するパッケージ番号の情報
Doc_index.htm	インテル ® MKL ドキュメントの目次
fftw2xmkl_notes.htm	FFTW 2.x インターフェイスのサポートに関するテクニカルノート
fftw3xmkl_notes.htm	FFTW 3.x インターフェイスのサポートに関するテクニカルノート
Install.txt	インストール・ガイド
mklman.pdf	インテル ® MKL リファレンス・マニュアル
mklman90_j.pdf	インテル ® MKL リファレンス・マニュアル (日本語)
Readme.txt	初期ユーザー情報
redist.txt	再配布可能ファイルのリスト
Release_Notes.htm	インテル ® MKL リリースノート (HTML 形式)
Release_Notes.txt	インテル ® MKL リリースノート (テキスト形式)
vmlnotes.htm	VML の一般的な説明
vslnotes.pdf	VSL の一般的な説明
userguide.pdf	インテル ® MKL Linux 版ユーザーズガイド (本ドキュメント)
./tables	vmlnotes.htm で参照されている表を含むディレクトリー

開発環境の構成

4

本章は、インテル ® マス・カーネル・ライブラリー (インテル ® MKL) を使用するために開発環境を構成する方法を説明します。また、特に、インテル ® MKL 構成ファイルを使用してカスタマイズ可能な機能について説明します。

スレッド化用に環境変数を設定する方法についての情報は、第 6 章の「OpenMP 環境変数を使用したスレッド数の設定」セクションを参照してください。

環境変数の設定

インテル ® MKL Linux* 版のインストールが完了すると、tools/environment ディレクトリーにある 3 つのスクリプト mklvars32、mklvarsem64t、および mklvars64 (.sh および .csh) を使用して、ユーザーシェルで環境変数 INCLUDE、LD_LIBRARY_PATH、MANPATH、CPATH、FPATH、および LIBRARY_PATH を設定できます。これらの変数をスタートアップ時に自動的に設定する方法は、「プロセスの自動化」セクションを参照してください。

インテル ® MKL の一部の機能は、構成ファイル mkl.cfg の変数を変更してカスタマイズできます。

プロセスの自動化

これらの環境変数をスタートアップ時に自動的に設定するには、ログインするたびに mklvars*.sh を実行するようにシェル・プロファイルを変更します。適切なインテル ® MKL ディレクトリーへのパスが設定されます。

ローカル・ユーザー・アカウントで以下のファイルを編集し、変数をエクスポートする直前のパス操作セクションで適切なスクリプトが実行されるようにコマンドを追加してください。追加するコマンドは次のようになります。

- bash:

  ~/.bash_profile、~/.bash_login または ~/.profile

  ```
  # bash の MKL 環境を設定します。
  . <MKL のインストール先への絶対パス>/tools/environment/mklvars<arch>.sh
  ```

- sh:

  ~/.profile

```
# sh の MKL 環境を設定します。
. <MKL のインストール先への絶対パス >/tools/environment/mklvars<arch>.sh
```

- csh:

```
~/.login

# csh の MKL 環境を設定します。
. <MKL のインストール先への絶対パス >/tools/environment/mklvars<arch>.csh
```

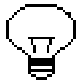
上記のコマンドで、`mklvars<arch>` は `mklvars32`、`mklvarsem64t` または `mklvars64` を表します。

スーパーユーザー権限がある場合、同じコマンドを `/etc/profile` (bash および sh の場合) または `/etc/csh.login` (csh の場合) のシステムファイルに追加してもかまいません。

ログイン中の問題を回避するため、インテル ® MKL をアンインストールする前に、スクリプトの実行を追加したすべてのプロファイル・ファイルから上記のコマンドを削除してください。

インテル ® MKL を Eclipse CDT でリンクする場合の構成

このセクションでは、インテル ® MKL を Eclipse C/C++ 開発ツール (CDT) 3.x および 4.0 でリンクする場合の構成を説明します。



ヒント : インテル ® MKL を CDT とリンクすると、Eclipse が提供する *コードアシスト* 機能を利用できます。*Eclipse ヘルプ* の *コード / コードアシスト* の説明を参照してください。

Eclipse CDT 4.0 の構成

インテル ® MKL を Eclipse CDT 4.0 でリンクするには、以下の手順を実行してください。

1. ツールチェイン / コンパイラーの統合でインクルード・パス・オプションをサポートしている場合、**[C/C++ General] > [Paths and Symbols]** プロパティー・ページの **[Includes]** タブを開いて、インテル ® MKL インクルード・パスを設定します。デフォルトは、`/opt/intel/mkl/10.0.xxx/include` (`xxx` はインテル ® MKL パッケージ番号、例えば `"039"`) です。
2. ツールチェイン / コンパイラーの統合でライブラリー・パス・オプションをサポートしている場合、**[C/C++ General] > [Paths and Symbols]** プロパティー・ページの **[Library Paths]** タブを開いて、ターゲットのアーキテクチャーに応じたインテル ® MKL ライブラリーのパスを設定します。デフォルトは、`/opt/intel/mkl/10.0.xxx/lib/em64t` です。

3. 特定のビルドでは、**[C/C++ Build]** > **[Settings]** プロパティー・ページの **[Tool Settings]** タブを開いて、アプリケーションとリンクするインテル ® MKL ライブラリーの名前 (例えば、`mkl_solver_lp64` および `mkl_core`) を指定します。ライブラリー・ファイルの名前ではなくライブラリー名を指定するコンパイラーでは、"`lib`" 接頭辞と "`a`" 拡張子は省略されます。ライブラリーの選択については、第 5 章の「リンクするライブラリーの選択」セクションを参照してください。ライブラリーを指定する際の項目設定の名前はコンパイラーの統合に依存します。

コンパイラー / リンカーは、自動メイクファイル生成が有効な場合のみ、インクルードとライブラリーのパス設定を自動的に使用することに注意してください。それ以外の場合、使用するメイクファイルにインクルードとライブラリーのパスを直接指定する必要があります。

Eclipse CDT 3.x の構成

インテル ® MKL を Eclipse CDT 3.x でリンクするには、以下の手順を実行してください。

- Standard Make プロジェクトの場合
  1. **[C/C++ Include Paths and Symbols]** プロパティー・ページを開いて、インテル ® MKL インクルード・パスを設定します。デフォルトは、
     `/opt/intel/mkl/10.0.xxx/include`
     (`xxx` はインテル ® MKL パッケージ番号、例えば "`039`") です。
  2. **[C/C++ Project Paths]** プロパティー・ページの **[Libraries]** タブを開いて、アプリケーションとリンクするインテル ® MKL ライブラリー (例えば、
     `/opt/intel/mkl/10.0.xxx/lib/em64t/libmkl_lapack.a` および
     `/opt/intel/mkl/10.0.xxx/lib/em64t/libmkl_em64t.a`) を設定します。ライブラリーの選択については、第 5 章の「リンクするライブラリーの選択」セクションを参照してください。

  標準メイクでは、上記の設定は CDT 内部関数のみに必要です。コンパイラー / リンカーはこれらの情報を自動的に取得しないため、メイクファイルで直接指定する必要があります。
- Managed Make プロジェクトの場合、特定のビルド用に設定を指定できます。次の操作を行ってください。
  1. **[C/C++ Build]** プロパティー・ページの **[Tool Settings]** タブを開きます。指定する必要のある設定はすべて、このページに含まれています。特定の設定の名前はコンパイラーの統合に依存するため、ここでは明記していません。
  2. コンパイラーの統合でインクルード・パス・オプションをサポートしている場合、インテル ® MKL インクルード・パスを設定します。デフォルトは、`/opt/intel/mkl/10.0.xxx/include` です。
  3. コンパイラーの統合でライブラリー・パス・オプションをサポートしている場合、ターゲットのアーキテクチャーに応じたインテル ® MKL ライブラリーのパスを設定します。デフォルトは、`/opt/intel/mkl/10.0.xxx/lib/em64t` です。
  4. アプリケーションとリンクするインテル ® MKL ライブラリーの名前 (例えば、`mkl_lapack` および `mkl_ia32`) を指定します。ライブラリー・ファイルの名前ではなくライブラリー名を指定するコンパイラーでは、"`lib`" 接頭辞と "`a`" 拡張子は省略されます。ライブラリーの選択については、第 5 章の「リンクするライブラリーの選択」セクションを参照してください。

5. インテル® MKL を使用するプロジェクトがアクティブになっていることを確認してください。

構成ファイルを使用したライブラリーのカスタマイズ

インテル® MKL 構成ファイルでは、ダイナミック・ライブラリーの名前を再定義できます。

ファイル名 `mkl.cfg` で構成ファイルを作成し、変数に値を割り当てることができます。次に、設定可能なすべての変数とデフォルト値を含む構成ファイルの例を示します。

例 4-1　　インテル® MKL 構成ファイル

```
//
// mkl.cfg ファイルのデフォルト値
//
// IA-32 アーキテクチャーの SO の名前
MKL_X87so = mkl_def.so
MKL_SSE1so = mkl_p3.so
MKL_SSE2so = mkl_p4.so
MKL_SSE3so = mkl_p4p.so
MKL_VML_X87so = mkl_vml_def.so
MKL_VML_SSE1so = mkl_vml_p3.so
MKL_VML_SSE2so = mkl_vml_p4.so
MKL_VML_SSE3so = mkl_vml_p4p.so
// インテル (R) 64 アーキテクチャーの SO の名前
MKL_EM64TDEFso = mkl_def.so
MKL_EM64TSSE3so = mkl_p4n.so
MKL_VML_EM64TDEFso = mkl_vml_def.so
MKL_VML_EM64TSSE3so = mkl_vml_p4n.so
// インテル (R) Itanium(R) プロセッサー・ファミリーの SO の名前
MKL_I2Pso = mkl_i2p.so
MKL_VML_I2Pso = mkl_vml_i2p.so
// LAPACK ライブラリーの SO の名前
MKL_LAPACKso = mkl_lapack.so
```

インテル® MKL 関数が呼び出されると、インテル® MKL は構成ファイルが存在するかどうかを確認し、存在する場合はファイルで指定された名前が使用されます。構成ファイルのパスは、`MKL_CFG_FILE` 環境変数で指定します。この変数が定義されていない場合、最初に現在のディレクトリー、次に `PATH` 環境変数で指定されたディレクトリーが検索されます。インテル® MKL 構成ファイルが存在しない場合、ライブラリーの標準名が使用されます。

変数が構成ファイルで指定されていない場合、または指定が正しくない場合も、ライブラリーの標準名が使用されます。

次に、ライブラリー名を再定義する構成ファイルの例を示します。

例 4-2　構成ファイルを使用したライブラリー名の再定義

```
// SO の再定義
MKL_X87so = matlab_x87.so
MKL_SSE1so = matlab_sse1.so
MKL_SSE2so = matlab_sse2.so
MKL_SSE3so = matlab_sse2.so
MKL_ITPso = matlab_ipt.so
MKL_I2Pso = matlab_i2p.so
```

アプリケーションとインテル ® マス・カーネル・ライブラリーのリンク

5

本章は、アプリケーションとインテル ® マス・カーネル・ライブラリー (インテル ® MKL) Linux* 版のリンクについて説明します。ここでは、スタティックとダイナミックのリンクモデルを比較します。インテル ® MKL ライブラリーのリンクに使用される一般的なリンク行の構文を説明します。また、特定のプラットフォームや関数領域用にアプリケーションとリンクするライブラリーの包括的な情報を表形式で提供します。さらに、リンクの例を提供します。また、カスタム共有オブジェクトの構築についても説明します。

リンクモデルの違い

アプリケーションとインテル ® MKL ライブラリーは、スタティック・ライブラリーまたは共有ライブラリーを使用して、スタティックまたはダイナミックにリンクできます。

スタティック・リンク

スタティック・リンクでは、リンクはすべてリンク時に解決されます。したがって、スタティックに構築された実行可能ファイルの動作は、そのファイルを実行するシステムで利用可能なライブラリーの特定のバージョンに依存しないため、完全に予測できます。これらの実行可能ファイルの動作は、テスト中の動作と完全に同じです。スタティック・リンクの主な短所は、スタティックにリンクしたアプリケーションのライブラリーを新しいバージョンにアップグレードする際、アプリケーション全体を再リンクする必要があるため、時間がかかることです。また、スタティック・リンクでは、複数の実行可能ファイルが同じライブラリーをリンクする場合、ライブラリーをメモリーに個別にロードするため、生成される実行可能ファイルのサイズが大きくなり、メモリーが効率的に使用されないことです。しかし、これは、主に大きなサイズの問題に使用されるインテル ® MKL ではほとんど問題になりません。実行可能ファイルのサイズと比較して相対的にデータのサイズが小さな実行可能ファイルでのみ問題になります。

ダイナミック・リンク

ダイナミック・リンク中、一部の未定義シンボルの解決はランタイムまで延期されます。ダイナミックに構築された実行可能ファイルには、未定義のシンボルとシンボルの定義を提供するライブラリーのリストが含まれます。実行可能ファイルをロードする際、最終的なリンクはアプリケーションが実行を開始する前に完了しています。複数のダイナミックに構築された実行可能ファイルが同じライブラリーを使用している場合、ライブラリーのメモリーへのロードは 1 回し

か行われず、共有して使用されるため、メモリーの節約になります。ダイナミックに構築されたアプリケーションはすべて同じライブラリーを共有するため、ダイナミック・リンクではライブラリーの使用とアップグレードの一貫性が保証されます。このリンク方法では、ライブラリーとライブラリーを使用するアプリケーションを別々に更新できるため、アプリケーションを最新に保つことが容易になります。ダイナミック・リンクでは、リンクの一部がランタイムに行われ、未解決のシンボルを専用のテーブルで確認して解決しなければならないため、ランタイム・パフォーマンスが多少損なわれます。しかし、インテル® MKL ではほとんど問題になりません。

リンクモデルの選択

アプリケーションを構築する際にインテル® MKL ライブラリーをダイナミックにリンクするかスタティックにリンクするかを選択するのはユーザーです。

ほとんどの場合、ユーザーはダイナミック・リンクを選択します。

しかし、サードパーティー向けのアプリケーションを開発していて、アプリケーション以外に何も送付しない場合、スタティック・リンクを使用する必要があります。送付される実行可能ファイルのサイズを減らすために、カスタム・ダイナミック・ライブラリー (「カスタム共有オブジェクトの構築」を参照) を構築することもできます。

表 5-1 は、リンクモデルの比較です。

表 5-1 インテル® MKL リンクモデルの比較

機能	ダイナミック・リンク	スタティック・リンク	カスタム・ダイナミック・リンク
プロセッサーのアップデート	自動	自動	再コンパイルして再配布
最適化	すべてのプロセッサー	すべてのプロセッサー	すべてのプロセッサー
ビルド	ダイナミック・ライブラリーにリンク	スタティック・ライブラリーにリンク	個別のダイナミック・ライブラリーをビルドしてリンク
呼び出し	通常名	通常名	修正名
合計バイナリーサイズ	大きい	小さい	小さい
実行可能ファイルのサイズ	最小	小さい	最小
マルチスレッド / スレッドセーフ	○	○	○

インテル® MKL 固有リンクの推奨

インテル® レガシー OpenMP* ランタイム・ライブラリー `libguide` およびインテル® 互換 OpenMP ランタイム・ライブラリー `libiomp` はダイナミックにリンクすることを強く推奨します。OpenMP ランタイム・ライブラリーへのスタティック・リンクは、ライブラリーの複数のコピーにソフトウェアがリンクされるため推奨しません。リンクすると、パフォーマンス問題 (スレッド数が多すぎる) が発生し、複数のコピーが初期化されるときに正当性問題が発生します。

ほかのライブラリーがスタティックにリンクされている場合でも、libguide および libiomp はダイナミックにリンクしてください。

リンクコマンドの構文

ファイル名が libyyy.a または libyyy.so のライブラリーをアプリケーションとリンクするには、2 つのオプションがあります。

- 以下のように、リンク行で、相対パスまたは絶対パスを使用してライブラリー・ファイルの名前をリストします。

  ```
  <ld> myprog.o /opt/intel/mkl/10.0.xxx/lib/32/libmkl_solver.a
  /opt/intel/mkl/10.0.xxx/lib/32/libmkl_intel.a
  /opt/intel/mkl/10.0.xxx/lib/32/libmkl_intel_thread.a
  /opt/intel/mkl/10.0.xxx/lib/32/libmkl_core.a
  /opt/intel/mkl/10.0.xxx/lib/32/libguide.so -lpthread
  ```

  <ld> はリンカー、myprog.o はユーザーのオブジェクト・ファイル、xxx はインテル ® MKL パッケージ番号 (例えば "039") です。
  適切なインテル ® MKL ライブラリーが最初にリストされ、システム・ライブラリー libpthread が続きます。
- リンク行で、-L*<path>* (バイナリーを検索する場所) および -I*<include>* (ヘッダーファイルを検索する場所) を指定してライブラリー名を (必要な場合、相対パスまたは絶対パスを含めて) リストします。インテル ® MKL ライブラリーとのリンクの説明では、このオプションを使用しています。

インテル ® MKL ライブラリーとリンクするには、以下の形式でパスとライブラリーを指定してください。

```
-L<MKL パス> -I<MKL インクルード>
[-lmkl_lapack95] [-lmkl_blas95]
[ クラスター・コンポーネント ]
[-Wl,--start-group]
[{-lmkl_{intel, intel_ilp64, intel_lp64, intel_sp2dp, gf, gf_ilp64, gf_lp64}]
[-lmkl_{intel_thread, sequential}]
[{-lmkl_solver, -lmkl_solver_lp64, -lmkl_solver_ilp64}]
{{[-lmkl_lapack] -lmkl_{ia32, em64t, ipf}},
-lmkl_core}}
[-Wl,--end-group]
[{-lguide, -liomp5}] [-lpthread] [-lm]
```

この構文の使用法の詳細およびインテル ® MKL の使用法に応じた推奨ライブラリーについては、「リンクするライブラリーの選択」を参照してください。以下の項目も参照してください。

- 第 7 章の「LAPACK と BLAS の Fortran 90 インターフェイスとラッパー」セクションのリンク前にビルドすべきライブラリーに関する情報

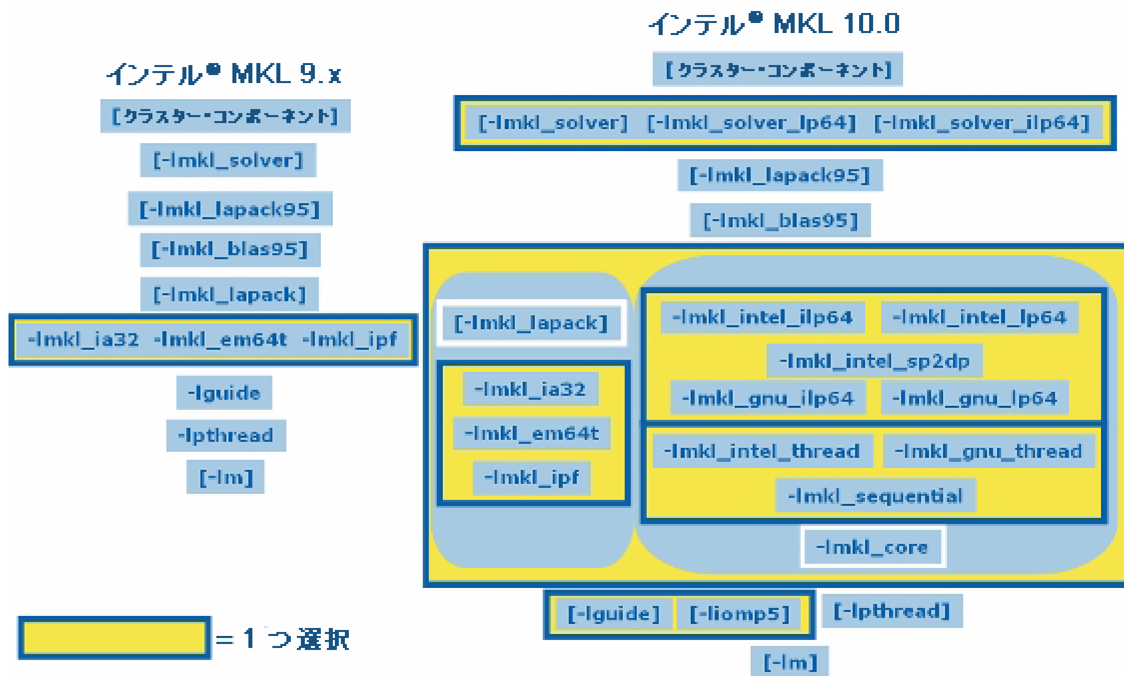
- 第 9 章「インテル® マス・カーネル・ライブラリー・クラスター・ソフトウェアの使用」のクラスター・コンポーネントの処理

インテル® MKL とリンクするには、リンク行で下位互換性がある*基本レイヤーモデル*または*デフォルト・レイヤー・モデル*を選択できます (クラスター・コンポーネントを除く)。上記の構文は両方のモデルを具体化したものです。

基本レイヤーモデルでは、インターフェイス・レイヤーから 1 つのライブラリー、スレッド化レイヤーから 1 つのライブラリー、計算レイヤーライブラリー (ライブラリーの選択は行いません) を選択して、ランタイム・ライブラリーを追加する必要があります。デフォルトモデルの場合、リンク行を変更する必要はありません (詳細は、第 3 章の「ダミー・ライブラリー」セクションを参照してください)。

図 5-1 は、レイヤーを使用するインテル® MKL 10.0 のリンクとインテル® MKL 9.x のリンクの比較です。

図 5-1　　　　インテル® MKL レイヤーとのリンク



スタティック・リンクで基本レイヤーモデルを使用する場合、インターフェイス・レイヤー、スレッド化レイヤー、および計算レイヤーバイナリーをグループ化シンボルで囲む必要があります (例えば、`-Wl,--start-group $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group`)。

ダミー・ライブラリーを使用する場合、

- インテル® MKL ライブラリーのパスは、リンカーがアーカイブ・ライブラリーを検索するパスの一覧に (例えば、-L<*MKL パス*> として) 追加する必要があります。

- インターフェイス・レイヤーまたはスレッド化レイヤーのライブラリーをリンク行に含めないでください。
- グループ化シンボルは使用しないでください。

リンク行でライブラリーをリストする順序は非常に重要です (グループ化シンボルで囲まれたライブラリーを除く)。

リンクするライブラリーの選択

次のリンク・ライブラリーの例は、インテル ® コンパイラー・インターフェイスを使用するコンポーネント別に、インテル ® 64 アーキテクチャー・ベースの 64 ビット Linux 用のデフォルト・レイヤー・リンク・モデルと基本レイヤー・リンク・モデルをリストしたものです。

- BLAS、FFT、VML、VSL コンポーネント、スタティック :

  デフォルトレイヤー : `libmkl_em64t.a`

  基本レイヤー : `libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`
- BLAS、FFT、VML、VSL コンポーネント、ダイナミック :

  デフォルトレイヤー : `libmkl.so`

  基本レイヤー : `libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_core.so`
- LAPACK、スタティック :

  デフォルトレイヤー : `libmkl_lapack.a libmkl_em64t.a`

  基本レイヤー : `libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`
- LAPACK、ダイナミック :

  デフォルトレイヤー : `libmkl_lapack.so libmkl.so`

  基本レイヤー : `libmkl_intel_lp64.so libmkl_intel_thread.so libmkl_core.so`
- ScaLAPACK、スタティック :

  デフォルトレイヤー : `libmkl_scalapack.a libmkl_blacs.a libmkl_lapack.a libmkl_em64t.a`

  基本レイヤー : `libmkl_intel_lp64.a libmkl_scalapack_core.a libmkl_blacs.a libmkl_intel_thread.a libmkl_core.a`
- PARDISO、スタティック :

  デフォルトレイヤー : `libmkl_solver.a libmkl_lapack.a libmkl_em64t.a`

  基本レイヤー、LP64: `libmkl_solver_lp64.a libmkl_intel_lp64.a libmkl_intel_thread.a libmkl_core.a`

  基本レイヤー、ILP64: `libmkl_solver_ilp64.a libmkl_intel_ilp64.a libmkl_intel_thread.a libmkl_core.a`

リンクする際は (「リンクコマンドの構文」および「その他のリンクの例」を参照)、次の点に注意してください。

- ソルバー・ライブラリーは現在、レイヤーモデルに対応していません。このため、インテル ® MKL 9.x に関する内部的な変更はありません。しかし、LP64/ILP64 インターフェイスをサポートするため、2 つのライブラリー (LP64 インターフェイス用の `libmkl_solver_lp64.a` と ILP64 インターフェイス用の `libmkl_solver_ilp64.a`)

が一体構造で導入されました。下位互換性のため、libmkl_solver.a はダミー・ライブラリーになりました。以前のリリース用のため、ソルバー・ライブラリーにはスタティック・バージョンしかありません。基本レイヤーモデルを使用してソルバー・ライブラリーとリンクするには、インターフェイスに応じて、リンク行で libmkl_solver_lp64.a または libmkl_solver_ilp64.a も指定する必要があります。

- libmkl_lapack95.a および libmkl_blas95.a ライブラリーにはそれぞれ、LAPACK95 および BLAS95 インターフェイスが含まれます。これらのライブラリーはオリジナルのディストリビューションには含まれておらず、インターフェイスを使用する前に構築する必要があります。(ライブラリーの構築についての詳細は第 7 章の「LAPACK と BLAS の Fortran 90 インターフェイスとラッパー」セクション、ソースコードが配布されている理由については「コンパイラー依存の関数と Fortran 90 モジュール」セクションを参照してください。)
- インテル® MKL FFT、VML、または VSL を使用する場合、リンク行に "-lm" を追加して Linux 数学サポート・ライブラリー libm をリンクする必要があります。
- Linux 版では、-lpthread を追加して pthreads ライブラリーをリンクする必要があります。pthread ライブラリーは Linux 固有で、libguide はこのライブラリーを使用してマルチスレッドをサポートします。libguide が必要な場合 (インテ® MKL からスレッド化されたソフトウェアを使用する場合)、リンク行の最後に -lpthread を追加する必要があります (リンクの順序は非常に重要です)。

スレッド化ライブラリーのリンク

これまで、インテル® コンパイラー以外にアプリケーションのスレッド化をサポートしているコンパイラーはごくわずかでした。現在は、いくつかのコンパイラーが OpenMP のスレッド化をサポートしています。アプリケーションがこれらのコンパイラーで OpenMP のスレッド化を使用してコンパイルされ、インテル® MKL の古いバージョン (9.x 以前) のスレッド化部分を呼び出した場合、問題が発生します。これは、MKL はインテル® コンパイラーを使用してスレッド化されていて、異なるコンパイラーのスレッド化ライブラリーとは互換性がないためです。同じアプリケーションで互換性のないスレッド化が使用された場合、エラーが発生することもあります。
インテル® MKL 10.0 では、この問題に対応するためにいくつかのソリューションを提供しています。これらのソリューションは、スレッド化レイヤーとコンパイラー・サポート RTL レイヤーのランタイム・ライブラリーで提供されます。

レイヤーのソリューション : インテル® MKL 10.0 では、ライブラリーはレイヤー構造になりました。これらのレイヤーの 1 つが、スレッド化レイヤーです。ライブラリーの内部構造のために、スレッド化はすべて、わずかなコードで表現されます。このコードは異なるコンパイラー (gnu コンパイラー など) でコンパイルされ、適切なレイヤーがスレッド化されたアプリケーションとリンクされます。

2 つめのコンポーネントは、コンパイラー・ポート RTL レイヤーです。インテル® MKL 9.x では、このレイヤーにはインテル® レガシー OpenMP ランタイム・コンパイラー・ライブラリー libguide のみが含まれていました。新しいバージョンでは、インテル® 互換 OpenMP ランタイム・ライブラリー libiomp を使用するという選択肢が追加されました。互換ライブラリーは、スレッド化コンパイラー (gnu) をサポートします。つまり、gnu コンパイラーを使用してスレッド化されたプログラムは、インテル® MKL および libiomp と安全にリンクし、効率的かつ効果的に実行することができます。

libiomp の詳細 : libiomp は新しいソフトウェアです。ベータ版からテストが行われてきましたが、不具合はほとんどありませんでした。さらに、コアの数が増加した際に、Microsoft や gnu スレッド化ライブラリーよりも優れたスケーリングを提供します。libiomp は、本質的に libguide スレッド管理ソフトウェアにコンパイラーが生成した関数呼び出しをマップするインターフェイス・レイヤーを含む libguide です。

表 5-2 は、使用されているスレッド化コンパイラーと状況別に、インテル ® MKL の現在のバージョンを使用した場合にスレッド化レイヤーと RTL レイヤーで選択するライブラリーを示したものです (スタティックのみ)。

表 5-2 スレッド化レイヤーと RTL レイヤーの選択

コンパイラー	マルチスレッド ?	スレッド化レイヤー	RTL レイヤーを推奨	コメント
Intel	問題になりません	mkl_intel_thread.a	libguide.so または libiomp5.so	
gnu	○	libmkl_gnu_thread.a	libiomp5.so または GNU OpenMP レイヤー	libiomp5 は優れたスケーリング・パフォーマンスを提供します
gnu	○	libmkl_sequential.a	なし	
gnu	X	libmkl_intel_thread.a	libguide.so または libiomp5.so	
その他	○	libmkl_sequential.a	なし	
その他	X	libmkl_intel_thread.a	libguide.so または libiomp5.so	

注 : アプリケーションをインテル ® MKL 9.x 以前の libguide とリンクした場合、インテル ® MKL 10.0 の libiomp は使用できません。

その他のリンクの例

以下に、インテル ® 64 アーキテクチャー・ベースのシステムでインテル ® コンパイラーを使用してリンクする場合の例をいくつか紹介します。これらの例では、*<MKL パス >* および *<MKL インクルード >* プレースホルダーは、ユーザーが定義した環境変数 $MKLPATH および $MKLINCLUDE に置換されています。第 9 章の ScaLAPACK およびクラスター FFT とのリンクの例も参照してください。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
```

```
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lguide -lpthread
```

ユーザーコード myprog.f と LP64 をサポートしているインテル® MKL の並列バージョンのスタティック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -lguide -lpthread
```

ユーザーコード myprog.f と LP64 をサポートしているインテル® MKL の並列バージョンのダイナミック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lpthread
```

ユーザーコード myprog.f と LP64 をサポートしているインテル® MKL の逐次バージョンのスタティック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread
```

ユーザーコード myprog.f と LP64 をサポートしているインテル® MKL の逐次バージョンのダイナミック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel_ilp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lguide -lpthread
```

ユーザーコード myprog.f と ILP64 をサポートしているインテル® MKL の並列バージョンのスタティック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_ilp64 -lmkl_intel_thread -lmkl_core -lguide -lpthread
```

ユーザーコード myprog.f と ILP64 をサポートしているインテル® MKL の並列バージョンのダイナミック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_lapack95
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lguide -lpthread
```

ユーザーコード myprog.f、Fortran 90 LAPACK インターフェイス[1]、LP64 をサポートしているインテル® MKL の並列バージョンのスタティック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_blas95
```

1. Fortran 90 LAPACK および BLAS インターフェイス・ライブラリーの構築方法については、第 7 章の「LAPACK と BLAS の Fortran 90 インターフェイスとラッパー」セクションを参照してください。

```
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lguide -lpthread
```

ユーザーコード myprog.f、Fortran 90 BLAS インターフェイス[1]、LP64 をサポートしているインテル ® MKL の並列バージョンのスタティック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -lmkl_solver_lp64.a
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lguide -lpthread
```

ユーザーコード myprog.f、スパースソルバーの並列バージョン、LP64 をサポートしているインテル ® MKL の並列バージョンのスタティック・リンク。

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_solver_lp64_sequential.a
-Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a -Wl,--end-group
-lpthread
```

ユーザーコード myprog.f、スパースソルバーの逐次バージョン、LP64 をサポートしているインテル ® MKL の逐次バージョンのスタティック・リンク。

他のリンク例は、インテル ® MKL サポート Web サイト http://www.intel.com/support/performancetools/libraries/mkl/ (英語) を参照してください。

リンクにおける注意

LD_LIBRARY_PATH の更新

インテル ® MKL 共有ライブラリーを使用する場合、共有ライブラリーのパス (LD_LIBRARY_PATH 環境変数) をライブラリーの場所に含めることを忘れないようにしてください。
インテル ® MKL ライブラリーが /opt/intel/mkl/10.0.xxx/lib/32 ディレクトリー (xxx はインテル ® MKL パッケージ番号、例えば "039") にある場合、bash シェルでコマンドラインは以下のようになります。

```
export LD_LIBRARY_PATH=/opt/intel/mkl/10.0.xxx/lib/32:$LD_LIBRARY_PATH
```

libguide のリンク

libguide をスタティックにリンクする場合 (非推奨) で、

- インテル ® コンパイラーを使用する場合、コンパイラーに含まれているバージョンの libguide を (-openmp オプションを使用して) リンクします。
- インテル ® コンパイラーを使用しない場合、インテル ® MKL に含まれているバージョンの libguide をリンクします。

スレッド化ライブラリーのダイナミック・リンク (libguide.so) を使用する場合 (推奨)、libguide の正しいバージョンが検索され、ランタイムに使用されるように、LD_LIBRARY_PATH が定義されていることを確認してください。

カスタム共有オブジェクトの構築

カスタム共有オブジェクトを使用すると、特定の問題を解くために必要な関数のコレクションをインテル ® MKL ライブラリーから減らすことができるため、ディスク容量が節約されます。また、独自のダイナミック・ライブラリーを構築できます。

インテル ® MKL カスタム共有オブジェクト・ビルダー

カスタム共有オブジェクト・ビルダーは、選択した関数を含むダイナミック・ライブラリー (共有オブジェクト) の作成をターゲットにします。ビルダーは、tools/builder ディレクトリーにあります。ビルダーには、メイクファイルおよび定義ファイル (関数のリスト) が含まれています。メイクファイルには、"ia32"、"ipf"、および "em64t" の 3 つのターゲットがあります。"ia32" は IA-32 アーキテクチャー対応プロセッサー、"ipf" は IA-64 アーキテクチャー対応プロセッサー、"em64t" はインテル ® 64 アーキテクチャー対応のインテル ® Xeon® プロセッサーに使用します。

メイクファイル・パラメーターの指定

メイクファイルには以下のパラメーター (マクロ) があります。

export = *functions_list*

共有オブジェクトに含まれるエントリーポイント関数のリストを含むファイルの名前を指定します。このファイルは定義ファイルの作成に使用された後、エクスポート・テーブルの作成に使用されます。デフォルト名は functions_list です。

name = *mkl_custom*

作成する ライブラリーの名前を指定します。デフォルトでは、ライブラリー mkl_custom.so が作成されます。

xerbla = *user_xerbla.o*

ユーザーのエラーハンドラーを含むオブジェクト・ファイルの名前を指定します。このエラーハンドラーは、ライブラリーに追加された後、標準 MKL エラーハンドラー xerbla の代わりに使用されます。デフォルトでは、このパラメーターは指定されないため、標準 MKL エラーハンドラー xerbla が使用されます。

すべてのパラメーターが必須ではありません。最も単純な場合、コマンドラインは make ia32 になります。残りのパラメーターの値はデフォルトになります。この結果、IA-32 アーキテクチャー対応プロセッサー用の mkl_custom.so ライブラリーが作成され、functions_list.txt ファイルの関数リスト、標準 MKL エラーハンドラー xerbla が使用されます。

以下は、より複雑な場合の別の例です。

```
make ia32 export=my_func_list.txt name=mkl_small xerbla=my_xerbla.o
```

この場合、IA-32 アーキテクチャー対応プロセッサー用の mkl_small.so ライブラリーが作成され、my_func_list.txt ファイルの関数リスト、およびユーザーのエラーハンドラー my_xerbla.o が使用されます。

インテル ® 64 または IA-64 アーキテクチャー対応プロセッサーの場合も処理はほぼ同様です。

関数のリストの指定

`functions_list` ファイルのエントリーポイントはインターフェイスに合わせて調整する必要があります。例えば、Fortran 関数は、ライブラリーに追加されるとき末尾に下線文字 "_" が追加されます。

```
dgemm_
ddot_
dgetrf_
```

選択した関数に複数のプロセッサー固有のバージョンがある場合、すべてのバージョンがカスタム・ライブラリーに追加され、ディスパッチャーによって管理されます。

パフォーマンスとメモリーの管理 6

本章は、インテル ® マス・カーネル・ライブラリー (インテル ® MKL) を使用して最適なパフォーマンスを得るための方法を説明します。まず、スレッド化 (「インテル ® MKL 並列処理の使用」を参照) について説明した後、ライブラリーのパフォーマンスを向上するためのコーディング・テクニックとハードウェア構成を示します。また、インテル ® MKL のメモリー管理と、デフォルトで使用するライブラリーのメモリー関数を再定義する方法についても説明します。

インテル ® MKL 並列処理の使用

インテル ® MKL は、直接法スパースソルバー、LAPACK (*GETRF、*POTRF、*GBTRF、*GEQRF、*ORMQR、*STEQR、*BDSQR、*SPTRF、*SPTRS、*HPTRF、*HPTRS、*PPTRF、*PPTRS ルーチン)、すべてのレベル 3 BLAS、圧縮形式のスパース列と対角形式のスパース BLAS 行列 - ベクトルおよび行列 - 行列乗算ルーチン、すべの FFT (DFTI_NUMBER_OF_TRANSFORMS=1 でサイズが 2 の累乗でない場合の 1D 変換を除く) でスレッド化されています。

注 : 1D FFT の 2 の累乗データについては、インテル ® MKL はサポートしている 3 つのアーキテクチャーすべてで並列処理を提供します。インテル ® 64 アーキテクチャーでは、並列処理は倍精度複素数のアウトオブプレース FFT でのみ提供されます。

ライブラリーは、OpenMP* スレッド化ソフトウェアを使用します。スレッド数の設定には、OMP_NUM_THREADS 環境変数を使用します。スレッド数を設定する手法は複数あることに注意してください。インテル ® MKL 10.0 よりも前のリリースでは、OMP_NUM_THREADS 環境変数 (詳細は、「OpenMP 環境変数を使用したスレッド数の設定」を参照) または等価な OpenMP ランタイム関数呼び出し (「ランタイムのスレッド数の変更」セクションを参照) を使用することができました。インテル ® MKL バージョン 10.0 から、MKL_NUM_THREADS のような OpenMP とは独立した変数と等価なインテル ® MKL 関数をスレッド管理に使用できるようになりました (詳細は、「新しいスレッド化コントロールの使用」を参照)。インテル ® MKL 変数は常に最初に検査され、次に OpenMP 変数が検査されます。どちらの変数も使用されていない場合、OpenMP ソフトウェアはデフォルトのスレッド数を選択します。インテル ® MKL 9.x 以前は、インテル ® コンパイラー OpenMP ソフトウェアのように、システムのプロセッサーの数と同じスレッド数がデフォルトのスレッド数として選択されていました。

注 : インテル ® MKL 10.0 では、OpenMP がデフォルトのスレッド数を決定します。

OMP_NUM_THREADS にはアプリケーションで使用するプロセッサーの数を常に設定することを推奨します。この設定は、異なる手法を使用して行うことができます。この後の「スレッド数を設定する手法」セクションを参照してください。

スレッド数を設定する手法

異なる手法を使用してインテル ® MKL で使用するスレッド数を指定することができます。

- OpenMP またはインテル ® MKL 環境変数を設定します。
  - OMP_NUM_THREADS
  - MKL_NUM_THREADS
  - MKL_DOMAIN_NUM_THREADS
- OpenMP またはインテル ® MKL 関数を呼び出します。
  - omp_set_num_threads()
  - mkl_set_num_threads()
  - mkl_domain_set_num_threads()

手法を選択する場合、以下の規則を考慮してください。

- 以前のインテル ® MKL バージョンのように、OpenMP 手法 (OMP_NUM_THREADS および omp_set_num_threads()) のみを使用している場合でも、ライブラリーは応答します。
- インテル ® MKL スレッド化コントロールは OpenMP 手法よりも優先されます。
- サブルーチン呼び出しは環境変数よりも優先されます。OpenMP サブルーチン omp_set_num_threads() の呼び出しは例外で、インテル ® MKL 環境変数 (MKL_NUM_THREADS など) が優先されます。
- 環境変数の読み取りは一度しか行われないため、ランタイム動作の変更には使用できません。

実行環境における競合の回避

インテル ® MKL でスレッドの使用が問題となる競合が存在する場合があります。ここでは、これらの問題への対処方法について説明します。最初に、なぜ問題が存在するかを簡単に説明します。

ユーザーが OpenMP 宣言子を使用してプログラムをスレッド化し、インテル ® コンパイラーを使用してプログラムをコンパイルした場合、そのプログラムとインテル ® MKL はどちらも同じスレッド化ライブラリーを使用します。インテル ® MKL はプログラムに並列領域が存在するかどうかを判断し、存在する場合、ユーザーが MKL_DYNAMIC 機能を使用してインテル ® MKL に指定しない限り、その処理を複数のスレッド上では行いません (詳細は「新しいスレッド化コントロールの使用」を参照)。しかし、インテル ® MKL が並列領域を認識できるのは、スレッド化プログラムとインテル ®

MKL が同じスレッド化ライブラリーを使用している場合のみです。ユーザーのプログラムがほかの手法でスレッド化されている場合、インテル® MKL はマルチスレッド・モードで動作しますが、リソースの浪費により適切なパフォーマンスが得られません。

使用するスレッド化モデル別に、推奨する競合の回避方法を説明します。

表 6-1 スレッド化モデル別の実行環境における競合の回避方法

スレッド化モデル	説明
OS スレッド (Linux* の場合 pthreads) を使用してプログラムをスレッド化する場合。	複数のスレッドがライブラリーを呼び出し、呼び出した関数がスレッド化されている場合、インテル® MKL のスレッド化をオフにする必要があります。利用可能な任意の手法を使用してスレッド数を設定します (「スレッド数を設定する手法」を参照)。
OpenMP 宣言子またはプラグマを使用してプログラムをスレッド化し、インテル以外のコンパイラーを使用してプログラムをコンパイルする場合。	OMP_NUM_THREADS 環境変数の設定がコンパイラーのスレッド化ライブラリーと libguide (libiomp) の両方に影響を与えるため、より問題です。この場合、インテル® MKL と使用する OpenMP コンパイラーのレイヤーが一致するように、スレッド化レイヤー・ライブラリーを選択してください (詳細は、「スレッド化ライブラリーのリンク」を参照)。選択できない場合、インテル® MKL の逐次バージョンをスレッド化レイヤーとして使用してください。この際、適切なスレッド化レイヤー・ライブラリー (libmkl_sequential.a または libmkl_sequential.so) をリンクする必要があります (第 3 章の「高レベル・ディレクトリー構造」セクションを参照)。
マルチ CPU システムで、各プロセッサーをノードとして扱い、コミュニケーションに MPI を使用する複数の並列プログラムが動作している場合。	各プロセッサーで個別の MPI プロセスが実行されていても、スレッド化ソフトウェアはシステムに複数のプロセッサーが存在することを認識します。この場合、利用可能な任意の手法を使用してスレッド数を設定します (「スレッド数を設定する手法」を参照)。

正当性とパフォーマンス問題を回避するには、インテル® レガシー OpenMP ランタイム・ライブラリー libguide およびインテル® 互換 OpenMP ランタイム･ライブラリー libiomp をダイナミックにリンクすることを強く推奨します。

OpenMP 環境変数を使用したスレッド数の設定

OMP_NUM_THREADS 環境変数を使用してスレッド数を設定することができます。スレッド数を変更するには、プログラムを実行するコマンドシェルで次のように入力します。

export OMP_NUM_THREADS=<*使用するスレッド数*> (bash などの特定のシェルの場合)

または

set OMP_NUM_THREADS=<*使用するスレッド数*> (csh や tcsh などのほかのシェルの場合)

インテル® MKL 環境変数 (例えば、MKL_NUM_THREADS) を使用してスレッド数を設定する方法は、「新しいスレッド化コントロールの使用」を参照してください。

ランタイムのスレッド数の変更

環境変数を使用してランタイムにプロセッサーの数を変更することはできません。しかし、プログラムから OpenMP API 関数を呼び出してランタイムにスレッドの数を変更することはできます。以下のサンプルコードでは、`omp_set_num_threads()` ルーチンを使用してランタイムにスレッドの数を変更しています。「スレッド数を設定する手法」も参照してください。

このサンプルを実行するには、インテル® コンパイラー・パッケージの `omp.h` ヘッダーファイルを使用します。インテル® コンパイラーがない場合、C バージョンではなく `omp_set_num_threads()` に Fortran API を使用してください。

例 6-1　　スレッド化用のプロセッサー数の変更

```
#include "omp.h"
#include "mkl.h"
#include <stdio.h>

#define SIZE 1000

void main(int args, char *argv[]){

   double *a, *b, *c;
   a = new double [SIZE*SIZE];
   b = new double [SIZE*SIZE];
   c = new double [SIZE*SIZE];

   double alpha=1, beta=1;
   int m=SIZE, n=SIZE, k=SIZE, lda=SIZE, ldb=SIZE, ldc=SIZE, i=0, j=0;
   char transa='n', transb='n';

   for( i=0; i<SIZE; i++){
       for( j=0; j<SIZE; j++){
           a[i*SIZE+j]= (double)(i+j);
           b[i*SIZE+j]= (double)(i*j);
           c[i*SIZE+j]= (double)0;
       }
   }
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                   m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
```

例 6-1　　　　スレッド化用のプロセッサー数の変更 (続き)

```
printf("row\ta\tc\n");
for ( i=0;i<10;i++){
    printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
}

omp_set_num_threads(1);

for( i=0; i<SIZE; i++){
    for( j=0; j<SIZE; j++){
        a[i*SIZE+j]= (double)(i+j);
        b[i*SIZE+j]= (double)(i*j);
        c[i*SIZE+j]= (double)0;
    }
}
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

 printf("row\ta\tc\n");
 for ( i=0;i<10;i++){
     printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
 }

 omp_set_num_threads(2);
 for( i=0; i<SIZE; i++){
     for( j=0; j<SIZE; j++){
         a[i*SIZE+j]= (double)(i+j);
         b[i*SIZE+j]= (double)(i*j);
         c[i*SIZE+j]= (double)0;
     }
 }
```

例 6-1 スレッド化用のプロセッサー数の変更 (続き)

```
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                   m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

    printf("row\ta\tc\n");
    for ( i=0;i<10;i++){
        printf("%d:\t%f\t%f\n", i, a[i*SIZE],
c[i*SIZE]);
    }

    delete [] a;
    delete [] b;
    delete [] c;
}
```

新しいスレッド化コントロールの使用

インテル® MKL 10.0 では、環境変数とサービス関数に新しいオプションのスレッド化コントロールが追加されています。これらのコントロールは OpenMP の等価な変数と似ていますが、OpenMP 変数よりも優先されます。これらのコントロールを OpenMP 変数とともに使用することで、インテル® MKL とライブラリーを互いに呼び出さないアプリケーションの部分をスレッド化することができます。

これらのコントロールを使用すると、OpenMP 設定とは無関係にインテル® MKL のスレッド数を指定できます。インテル® MKL は実際には推奨と異なるスレッド数を使用することがありますが、コントロールはライブラリーを呼び出すアプリケーションの検出できないスレッド化動作のイベントで、推奨するスレッド数を使用してみるようにライブラリーに指示します。

注 : インテル® MKL では、システムリソースなどの特定の理由により、スレッド数を常に選択できるとは限りません。

アプリケーションでインテル® MKL スレッド化コントロールを使用するかどうかは任意です。コントロールを使用しない場合、ライブラリーはデフォルトのスレッド数が異なることを除けば、スレッド化に関してインテル® MKL 9.1 と同じように動作します。使用法の違いは、「FFT の使用法に関する注意」を参照してください。

表 6-2 は、スレッド化コントロール用のインテル® MKL 環境変数とサービス関数、および等価な OMP 環境変数の一覧です。

表 6-2　　スレッド化コントロール用のインテル ® MKL 環境変数

環境変数	サービス関数	内容	等価 OMP 環境変数
MKL_NUM_THREADS	mkl_set_num_threads	使用するスレッド数を示します。	OMP_NUM_THREADS
MKL_DOMAIN_NUM_THREADS	mkl_domain_set_num_threads	特定の関数領域用のスレッド数を示します。	
MKL_DYNAMIC	mkl_set_dynamic	インテル ® MKL がスレッド数を動的に変更できるようにします。	OMP_DYNAMIC

注 : 関数はそれぞれの環境変数よりも優先されます。
特に、アプリケーションでインテル ® MKL が指定されたスレッド数を使用し、アプリケーションのユーザーが環境変数を使用してスレッド数を変更しないようにするには、mkl_set_num_threads() を呼び出してスレッド数を設定します。この設定は、環境変数の設定よりも優先されます。

次の例は、インテル ® MKL 関数 mkl_set_num_threads() を使用してインテル ® MKL 9.x のデフォルトの動作 (1 つのスレッドで実行) を行う方法を示しています。

例 6-2　　スレッド数を 1 に設定

```
#include <omp.h>
#include <mkl.h>
…
mkl_set_num_threads ( 1 );
```

この後のセクションで、スレッド化コントロール用のインテル ® MKL 環境変数について詳細に説明します。スレッド化コントロール関数、パラメーター、呼び出し構文、コードの例は、*『インテル ® MKL リファレンス・マニュアル』*を参照してください。

MKL_DYNAMIC

MKL_DYNAMIC のデフォルト値は、OMP_DYNAMIC のデフォルト値が FALSE の場合でも、TRUE に設定されます。

MKL_DYNAMIC が TRUE の場合、インテル ® MKL は常に、ユーザーが指定した最大値までの、最良とみなすスレッド数を選択します。MKL_DYNAMIC が FALSE の場合、インテル ® MKL は、特に理由がない限り、ユーザーが指定したスレッド数を使用します。

MKL_DYNAMIC=FALSE の場合でも、インテル® MKL が指定したスレッド数を使用することは保証されていないことに注意してください。ライブラリーは問題を調査し、推奨値と異なるスレッド数を使用することがあります。例えば、8 つのスレッドでサイズ 1 の行列 - 行列乗算を行おうとすると、このイベントで 8 つのスレッドを使用することは実用的でないため、ライブラリーは代わりに 1 スレッドのみを使用します。

インテル® MKL が並列領域で呼び出された場合、デフォルトでは 1 つのスレッドのみを使用することにも注意してください。ライブラリーで入れ子の並列処理を使用し、並列領域内のスレッドをインテル® MKL が使用しているものと同じ OpenMP コンパイラーでコンパイルする場合、MKL_DYNAMIC を FALSE に設定し、スレッド数を手動で設定してください。

一般に、ライブラリーが並列セクションからすでに呼び出されていて入れ子の並列処理が望ましい場合など、インテル® MKL が検出できない状況でのみ、MKL_DYNAMIC を FALSE に設定してください。

MKL_DOMAIN_NUM_THREADS

MKL_DOMAIN_NUM_THREADS には、文字列値 *<MKL 環境文字列＞*を以下の形式で指定します。

<MKL 環境文字列＞ ::= <MKL 領域環境文字列＞ { ＜区切り文字＞ <MKL 領域環境文字列＞ }

＜区切り文字＞ ::= [＜スペース記号＞] (＜スペース記号＞ | ＜カンマ記号＞ | ＜セミコロン記号＞ | ＜コロン記号＞) [＜スペース記号＞*]*

<MKL 領域環境文字列＞ ::= <MKL 領域環境名＞ ＜使用法＞ ＜スレッド数＞

<MKL 領域環境名＞ ::= MKL_ALL | MKL_BLAS | MKL_FFT | MKL_VML

＜使用法＞ ::= [＜スペース記号＞] (＜スペース記号＞ | ＜等号記号＞ | ＜カンマ記号＞) [＜スペース記号＞*]*

＜スレッド数＞ ::= ＜正の整数＞

＜正の整数＞ ::= <10 進数＞ | <8 進数＞ | <16 進数＞

上記の構文で、MKL_BLAS は BLAS 関数領域、MKL_FFT は非クラスター FFT、MKL_VML はベクトル数学ライブラリーを示します。

例 :

```
MKL_ALL 2 : MKL_BLAS 1 : MKL_FFT 4

MKL_ALL=2 : MKL_BLAS=1 : MKL_FFT=4

MKL_ALL=2,  MKL_BLAS=1,  MKL_FFT=4

MKL_ALL=2;  MKL_BLAS=1;  MKL_FFT=4

MKL_ALL   = 2  MKL_BLAS 1 ,  MKL_FFT  4

MKL_ALL,2: MKL_BLAS 1, MKL_FFT,4 .
```

グローバル変数 MKL_ALL、MKL_BLAS、MKL_FFT、MKL_VML、およびインテル® MKL スレッド化コントロール用のインターフェイスは、mkl.h ヘッダーファイルに記述されています。

表 6-3 は、MKL_DOMAIN_NUM_THREADS の値がどのように解釈されるかを示しています。

表 6-3 MKL_DOMAIN_NUM_THREADS の値の解釈

MKL_DOMAIN_NUM_THREADS の値	解釈
MKL_ALL=4	インテル® MKL のすべての部分で 4 つのスレッドを使用するように推奨します。実際のスレッド数は、MKL_DYNAMIC の設定やシステムリソースの状況に応じて異なります。この設定は、MKL_NUM_THREADS = 4 と等価です。
MKL_ALL=1, MKL_BLAS=4	BLAS で 4 つのスレッドを使用することを除いて、インテル® MKL の残りの部分で 1 つのスレッドを使用するように推奨します。
MKL_VML = 2	VML で 2 つのスレッドを使用するように推奨します。設定は、インテル® MKL のほかの部分には影響しません。

注 : 領域固有の設定は、ほかの設定よりも優先されます。例えば、MKL_DOMAIN_NUM_THREADS が "MKL_BLAS=4" に設定された場合、MKL_NUM_THREADS の設定に関係なく、BLAS で 4 つのスレッドを使用するように推奨します。関数呼び出し "mkl_domain_set_num_threads (4, MKL_BLAS);" も、mkl_set_num_threads() への呼び出しに関係なく、BLAS で 4 つのスレッドを使用するように推奨します。
しかし、"mkl_domain_set_num_threads (4, MKL_ALL);" のように、"MKL_ALL" が入力の関数呼び出しは、"mkl_set_num_threads(4)" と等価であるため、後の mkl_set_num_threads 呼び出しよりも優先されることに注意してください。同様に、MKL_DOMAIN_NUM_THREADS が "MKL_ALL=4" に設定された場合、MKL_NUM_THREADS = 2 よりも優先されます。

例えば、MKL_DOMAIN_NUM_THREADS 環境変数では "MKL_BLAS=4,MKL_FFT=2" のように複数の変数を一度に設定することができますが、対応する関数で文字列構文は使用できません。このため、関数呼び出しで同じことを行うには、以下のように複数の呼び出しを行う必要があります。

```
mkl_domain_set_num_threads ( 4, MKL_BLAS );
mkl_domain_set_num_threads ( 2, MKL_FFT );
```

スレッド化コントロール用の環境変数の設定

スレッド化コントロールに使用する環境変数を設定するには、プログラムを実行するコマンドシェルで次のように入力します。

export <*変数名*>=<*値*> (bash などの特定のシェルの場合)

例 :

```
export MKL_NUM_THREADS=4
export MKL_DOMAIN_NUM_THREADS="MKL_ALL=1, MKL_BLAS=4"
export MKL_DYNAMIC=FALSE
```

csh や tcsh などの他のシェルの場合は、次のように入力します。

set *<変数名>*=*<値>*

例 :

```
set MKL_NUM_THREADS=4
set MKL_DOMAIN_NUM_THREADS="MKL_ALL=1, MKL_BLAS=4"
set MKL_DYNAMIC=FALSE
```

FFT の使用法に関する注意

新しいスレッド化コントロールの追加により、FFT 実装の遂行段階が最適化され、倍精度データの初期化を回避できるようになりました。しかし、この最適化を使用するには FFT の使用法を変更する必要があります。例えば、FFT ディスクリプターをすべて初期化した後でアプリケーションでスレッドを作成すると仮定します。この場合、スレッド化は並列 FFT 計算でのみ行われ、ディスクリプターは並列領域から戻った後に解放されます。各ディスクリプターは対応するスレッド内でのみ使用されます。インテル® MKL 10.0 では、遂行段階の前に 1 つのスレッドで動作するように、ライブラリーに明示的に指示する必要があります。このためには、MKL_NUM_THREADS=1 または MKL_DOMAIN_NUM_THREADS="MKL_FFT=1" を設定するか、サービス関数の対応するペアを呼び出します。指示を行わないと、DftiCommitDescriptor 関数が並列領域にないために実際のスレッド数が異なることがあります。*『インテル® MKL リファレンス・マニュアル』の「例 C-27a: 1 つのスレッドで初期化された複数のディスクリプターを伴う並列モードの使用」*を参照してください。

パフォーマンスを向上するためのヒントと手法

インテル® MKL を使用して最適なパフォーマンスを得るには、この後に説明されている推奨事項に従ってください。

コーディング手法

インテル® MKL を使用して最適なパフォーマンスを得るため、ソースコードでデータが以下のようにアライメントされていることを確認してください。

- 配列が 16 バイト境界でアライメントされている
- 2 次元配列のリーディング・ディメンジョンの値 (n*element_size) が 16 の倍数である
- 2 次元配列のリーディング・ディメンジョンの値が 2048 の倍数でない

LAPACK 圧縮ルーチン

名前の行列の型と格納位置 (2 つめと 3 つめの文字) が HP、OP、PP、SP、TP、UP のルーチンは、圧縮形式で行列を処理します (*『インテル® MKL リファレンス・マニュアル』*の LAPACK の *「ルーチン命名規則」*セクションを参照)。これらの機能は、名前の行列の型と格納位置 (2 つめと 3 つめの文字) が HE、OR、PO、SY、TR、UN の非圧縮ルーチンの機能と厳密に等価ですが、パフォーマンスは大幅に低くなります。

メモリー制限があまり厳しくない場合は、非圧縮ルーチンを使用してください。この場合、それぞれの圧縮ルーチンで要求されるメモリーよりも $N^2/2$ 多いメモリーを割り当てる必要があります。*N* は問題サイズ (方程式の数) です。

例えば、高度ドライバーを使用して対称固有値問題を解く場合、次の非圧縮ルーチンを使用することで高速化が期待できます。

```
call dsyevx(jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m,
w, z, ldz, work, lwork, iwork, ifail, info)
```

a は少なくとも N^2 の要素を含む次元 *lda* × *n* です。変更前は次のとおりです。

```
call dspevx(jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m, w, z,
ldz, work, iwork, ifail, info)
```

ap は次元 *N**(*N*+1)/2 です。

FFT 関数

FFT 関数のパフォーマンスを向上する追加の条件があります。

IA-32 またはインテル ® 64 アーキテクチャー・ベースのアプリケーション : 2 次元配列の最初の要素のアドレスとリーディング・ディメンジョンの値 (`n*element_size`) が、以下のキャッシュ・ライン・サイズの倍数である必要があります。

- 32 バイト (インテル ® Pentium® III プロセッサーの場合)
- 64 バイト (インテル ® Pentium® 4 プロセッサーの場合)
- 128 バイト (インテル ® 64 アーキテクチャー対応プロセッサーの場合)

IA-64 アーキテクチャー・ベースのアプリケーション : 十分条件は次のとおりです。

- C 形式の FFT では、実数部と虚数部を表す配列間の距離 *L* が 64 の倍数でない - 最良のケースは *L*=*k**64 + 16
- 2 次元配列のリーディング・ディメンジョンの値 (`n*element_size`) が 2 の累乗でない

ハードウェア構成のヒント

デュアルコア インテル ® Xeon® プロセッサー 5100 番台のシステム :
デュアルコア インテル ® Xeon® プロセッサー 5100 番台のシステムで最良のインテル ® MKL パフォーマンスを得るには、このプロセッサーの*ハードウェア DPL (ストリーミング・データ) プリフェッチャー*機能を有効にしてください。この機能を有効にするには、適切な BIOS 設定を行う必要があります。詳細は、BIOS のドキュメントを確認してください。

ハイパースレッディング・テクノロジーの使用 : ハイパースレッディング・テクノロジー (HT テクノロジー) は、各スレッドが異なる演算を実行している場合、またはプロセッサー上に十分に活用されていないリソースがある場合に特に有効です。インテル ® MKL は、このどちらにもあてはまりません。ライブラリーのスレッド化された部分が効率的に (利用可能なリソースの大半を使用して) 実行され、各スレッドで同一の演算を行っているためです。*HT テクノロジーを有効にせずに*インテル ® MKL を使用すると、より高いパフォーマンスを得られることがあります。

マルチコア・パフォーマンスの管理

スレッドを処理するコアが変更されないようにすることで、マルチコア・プロセッサーのシステムで最良のパフォーマンスを得ることができます。このためには、スレッドにアフィニティー・マスクを設定し、スレッドと CPU コアをバインドします。利用可能な場合、OpenMP の機能 (例えば、インテル® OpenMP を使用して `KMP_AFFINITY` 環境変数で設定) を使用することを推奨します。または、以下の例のようにシステムルーチンを使用します。

以下の仮定を行います。

- システムにそれぞれ 2 つのコアがある 2 つのソケットがある
- インテル® MKL FFT を呼び出す 2 スレッドの並列アプリケーションが 4 スレッドで高速に実行されるが、2 スレッドのパフォーマンスが非常に不安定

この場合、

1. 異なるソケットのコアにスレッドをバインドする FFT 呼び出しの前に、以下のコードを追加します。
2. アプリケーションをビルドして、2 スレッドで実行します。

```
env OMP_NUM_THREADS=2 ./a.out
```

例 6-3　オペレーティング・システムでアフィニティー・マスクを設定してインテル® コンパイラーを使用

```
// アフィニティー・マスクを設定します。
#include <sched.h>
#include <omp.h>
#pragma omp parallel default(shared)
{
   unsigned long mask = (1 << omp_get_thread_num()) * 2;
   sched_setaffinity( 0, sizeof(mask), &mask );
     }
// MKL FFT ルーチンを呼び出します。
```

上記の例で使用されている `sched_setaffinity` `SetThreadAffinityMask` 関数の詳細については、*Linux プログラマーズ・マニュアル* (man ページ形式) を参照してください。

非正規化数の演算

インテル® MKL 関数が非正規化数 (指定された浮動小数点形式でサポートされている最も小さな非ゼロの数よりも小さな非ゼロの数) を演算するか、計算中に非正規化数が算出された場合 (例えば、処理するデータがアンダーフローのしきい値に非常に近い場合)、パフォーマンスが低下することが考えられます。非正規化数の浮動小数点演算を行うと、例外ハンドラーを呼び出すように CPU ステートが設定され、アプリケーションが遅くなります。

この問題を解決するには、-ftz オプションを使用して (インテル ® コンパイラーの場合) メインプログラムをコンパイルします。このオプションを使用すると、非正規化数はプロセッサー・レベルでゼロとして扱われ、例外ハンドラーは呼び出されません。このオプションを設定すると、精度に多少影響することに注意してください。

この問題を解決する別の方法は、アンダーフローのしきい値に近い値を避けるように入力データの適切なスケーリングを行う方法です。

FFT 最適化基数

データベクトルの長さが最適化基数の累乗に因数分解できる場合、インテル ® MKL FFT の最適なパフォーマンスを得ることができます。

インテル ® MKL では、最適化基数の一覧はアーキテクチャーに依存します。

- 2, 3, 4, 5 (IA-32 アーキテクチャー)
- 2, 3, 4, 5 (インテル ® 64 アーキテクチャー)
- 2, 3, 4, 5, 7, 11 (IA-64 アーキテクチャー)

インテル ® MKL メモリー管理の使用

インテル ® MKL には、ライブラリー関数によって使用されるメモリーバッファーを管理するメモリー管理ソフトウェアが用意されています。特定の関数 (レベル 3 BLAS または FFT) が呼び出されるときにライブラリーが割り当てる新しいバッファーは、プログラムが終了するまで解放されません。ある地点でプログラムがメモリーを解放する必要がある場合、MKL_FreeBuffers() を呼び出します。メモリーバッファーが必要なライブラリー関数に別の呼び出しが行われると、メモリー・マネージャーはバッファーを再び割り当てます。このバッファーは、プログラムが終了するかプログラムがメモリーを解放するまで割り当てられたままです。

この動作により、パフォーマンスが向上します。しかし、いくつかのツールではこの動作をメモリーリークとして報告することがあります。インテル ® MKL で利用可能な関数を使用してプログラムのメモリーを解放することができます。また、環境変数を設定して各呼び出しの後にメモリーを解放することもできます。

メモリー管理ソフトウェアはデフォルトで有効になっています。環境変数を使用してソフトウェアを無効にするには、MKL_DISABLE_FAST_MM に任意の値を設定してください。メモリーは呼び出しごとに割り当てられ呼び出しの後に解放されます。この機能を無効にすると、特に問題サイズの小さな、レベル 3 BLAS などのルーチンのパフォーマンスが低下します。

しかし、これらの方法を使用してメモリーを解放しても、メモリーリークが報告されなくなるとは限りません。実際、ライブラリーを複数回呼び出す場合、各呼び出しごとに新しいメモリーの割り当てが必要になり、報告される数は増えることもあります。上記の方法で解放されなかったメモリーは、プログラムの終了時にシステムによって解放されます。

各スレッドに割り当てることのできるバッファーの数には制限があります。現在、この数は 32 です。サポートしているスレッドの最大数は 514 です。デフォルトの制限を回避するには、メモリー管理を無効にしてください。

メモリー関数の再定義

インテル® MKL 9.0 では、ライブラリーがデフォルトで使用するメモリー関数を独自の関数に置換できるようになりました。この処理は、*メモリー関数名の変更*機能を使用して行います。

メモリー関数名の変更

一般に、ユーザーが同様のシステム関数 (malloc、free、calloc、および realloc) の代わりに独自のメモリー管理関数を使用しようとすると、メモリーが 2 つの独立したメモリー管理パッケージによって管理されるため、メモリー問題が発生します。この問題を防ぐために、特定のインテル® ライブラリーとインテル® MKL の特定の項目にメモリー関数名の変更機能が追加されました。この機能を使用すると、ユーザーはメモリー管理関数を再定義することができます。

これは、インテル® MKL は、システム関数ではなくメモリー関数へのポインター (i_malloc、i_free、i_calloc、i_realloc) を実際に使用するためです。これらのポインターは最初はシステムのメモリー管理関数 (malloc、free、calloc、realloc) のアドレスを保持しアプリケーション・レベルで見ることができます。このため、ポインター値をプログラムで再定義することが可能です。

ユーザーがこれらのポインターをユーザー定義のメモリー管理関数へリダイレクトすると、メモリーはシステム関数ではなくユーザー定義関数で管理されます。1 つの (ユーザー定義の) メモリー管理パッケージだけが使用されれば、問題は回避されます。

デフォルトでは、インテル® MKL メモリー管理は、標準 C ランタイムメモリー関数を使用してメモリーの割り当てと解放を行います。これらの関数はメモリー関数名の変更機能を使用して置換することができます。

メモリー関数の再定義方法

メモリー関数を再定義するには、以下の操作を行ってください。

1. i_malloc.h ヘッダーファイルをコードにインクルードします。
   (ヘッダーファイルには、アプリケーション開発者がメモリー割り当て関数を置換するために必要な宣言がすべて含まれています。このヘッダーファイルでは、この機能をサポートするインテル® ライブラリーでメモリー割り当てを置換する方法も説明されています。)
2. インテル® MKL 関数への最初の呼び出しの前にポインター i_malloc、i_free、i_calloc、i_realloc の値を再定義します。

例 6-4　　メモリー関数の再定義

```
#include "i_malloc.h"
   .  .  .
   i_malloc  = my_malloc;
   i_calloc  = my_calloc;
   i_realloc = my_realloc;
   i_free    = my_free;
   . . .
// ここでインテル® MKL 関数を呼び出すことができます。
```

言語固有の使用法オプション 7

インテル ® マス・カーネル・ライブラリー (インテル ® MKL) は、Fortran と C/C++ プログラミングを基本的にサポートしています。しかし、すべての関数領域で Fortran と C インターフェイスの両方をサポートしているとは限りません (「表 A-1」を参照)。例えば、LAPACK には C インターフェイスはありません。しかし、混在言語プログラミングを使用して C からこれらの領域を含む関数を呼び出すことはできます。

Fortran 90 環境で、基本的に Fortran をサポートする LAPACK や BLAS を使用する場合でも、ソースコードで提供されている言語固有のインターフェイス・ライブラリーとモジュールをビルドする作業が最初に必要になります。

本章は、主に混在言語プログラミングと言語固有のインターフェイスの使用について説明します。基本的に Fortran をサポートする関数領域で C 言語環境のインテル ® MKL を使用する方法と、言語固有のインターフェイス、特に Fortran 90 インターフェイスを LAPACK および BLAS に使用する方法を説明します。なぜ Fortran 90 のモジュールがソースで提供されているかを説明するため、コンパイラー依存の関数について説明します。別のセクションでは、Java からインテル ® MKL 関数を呼び出すサンプルを実行する過程を順を追って説明します。

言語固有インターフェイスとインテル ® MKL の使用

interfaces ディレクトリーにあるメイクファイルを実行すると、以下のインターフェイス・ライブラリーとモジュールが生成されます。

表 7-1　インターフェイス・ライブラリーとモジュール

ファイル名	内容
libmkl_blas95.a	BLAS (BLAS95) 用 Fortran 90 ラッパー
libmkl_lapack95.a	LAPACK (LAPACK95) 用 Fortran 90 ラッパー
libfftw2xc_intel.a	インテル ® MKL FFT を呼び出す FFTW バージョン 2.x 用インターフェイス (インテル ® コンパイラー用 C インターフェイス)
libfftw2xc_gnu.a	インテル ® MKL FFT を呼び出す FFTW バージョン 2.x 用インターフェイス (GNU コンパイラー用 C インターフェイス)
libfftw2xf_intel.a	インテル ® MKL FFT を呼び出す FFTW バージョン 2.x 用インターフェイス (インテル ® コンパイラー用 Fortran インターフェイス)
libfftw2xf_gnu.a	インテル ® MKL FFT を呼び出す FFTW バージョン 2.x 用インターフェイス (GNU コンパイラー用 Fortran インターフェイス)

表 7-1 インターフェイス・ライブラリーとモジュール (続き)

ファイル名	内容
libfftw3xc_intel.a	インテル ® MKL FFT を呼び出す FFTW バージョン 3.x 用インターフェイス (インテル ® コンパイラー用 C インターフェイス)
libfftw3xc_gnu.a	インテル ® MKL FFT を呼び出す FFTW バージョン 3.x 用インターフェイス (GNU コンパイラー用 C インターフェイス)
libfftw3xf_intel.a	インテル ® MKL FFT を呼び出す FFTW バージョン 3.x 用インターフェイス (インテル ® コンパイラー用 Fortran インターフェイス)
libfftw3xf_gnu.a	インテル ® MKL FFT を呼び出す FFTW バージョン 3.x 用インターフェイス (GNU コンパイラー用 Fortran インターフェイス)
libfftw2x_cdft_SINGLE.a	インテル ® MKL クラスター FFT を呼び出す MPI FFTW バージョン 2.x 用単精度インターフェイス (C インターフェイス)
libfftw2x_cdft_DOUBLE.a	インテル ® MKL クラスター FFT を呼び出す MPI FFTW バージョン 2.x 用倍精度インターフェイス (C インターフェイス)
mkl95_blas.mod	BLAS (BLAS95) 用 Fortran 90 インターフェイス・モジュール
mkl95_lapack.mod	LAPACK (LAPACK95) 用 Fortran 90 インターフェイス・モジュール
mkl95_precision.mod	BLAS95 および LAPACK95 用精度パラメーターの Fortran 90 定義

「LAPACK と BLAS の Fortran 90 インターフェイスとラッパー」セクションでは、これらのライブラリーとモジュールがどのように生成されるかを例で示しています。

LAPACK と BLAS の Fortran 90 インターフェイスとラッパー

Fortran 90 インターフェイスは基本プロシージャー用に提供されています。ラッパーはソースで提供されます。(詳細は、「コンパイラー依存の関数と Fortran 90 モジュール」を参照)。これらのインターフェイスを使用する最も簡単な方法は、対応するライブラリーをビルドしてユーザーのライブラリーとしてリンクする方法です。この操作を行うには、管理者権限が必要です。提供されている製品ディレクトリーに書き込み可能であれば、操作は単純です。

1. `mkl/10.0.xxx/interfaces/blas95` または `mkl/10.0.xxx/interfaces/lapack95` ディレクトリーを開きます。`xxx` はインテル ® MKL パッケージ番号 (例えば "`039`") です。
2. 以下のいずれかのコマンドを入力します。

   `make PLAT=lnx32 lib` - (IA-32 アーキテクチャー)

   `make PLAT=lnx32e lib` - (インテル ® 64 アーキテクチャー)

   `make PLAT=lnx64 lib` - (IA-64 アーキテクチャー)

必要なライブラリーと `.mod` ファイルがビルドされ、リリースの標準カタログにインストールされます。

`.mod` ファイルは、次のコンパイラー・コマンドを使用してインターフェイスのファイルから取得することもできます。

`ifort -c mkl_lapack.f90` または `ifort -c mkl_blas.f90`

これらのファイルは、*`include`* ディレクトリーにあります。

管理者権限がない場合、次の操作を行ってください。

1. ディレクトリー (mkl/10.0.xxx/interfaces/blas95 または mkl/10.0.xxx/interfaces/lapack95) 全体をユーザー定義ディレクトリー *<user_dir>* にコピーします。
2. 対応するファイル (mkl_blas.f90 または mkl_lapack.f90) を mkl/10.0.xxx/include からユーザー定義ディレクトリー *<user_dir>*/blas95 または *<user_dir>*/lapack95 にコピーします。
3. 上記の make コマンドの 1 つを *<user_dir>*/blas95 または *<user_dir>*/lapack95 ディレクトリーで変数を指定して実行します。

```
make PLAT=lnx32 INTERFACE=mkl_blas.f90 lib
make PLAT=lnx32 INTERFACE=mkl_lapack.f90 lib
```

必要なライブラリーと .mod ファイルがビルドされ、*<user_dir>*/blas95 または *<user_dir>*/lapack95 ディレクトリーにインストールされます。

デフォルトでは、ifort コンパイラーが選択されています。make の FC=*<compiler>* パラメーターを使用してコンパイラーを変更することもできます。

例 :

```
make PLAT=lnx64 FC=<compiler> lib
```

ライブラリーをビルドしないでインターフェイスを使用する別の方法もあります。

ビルド用ディレクトリーからライブラリーを削除するには、以下のコマンドを使用します。

make PLAT=lnx32 clean	- (IA-32 アーキテクチャー)
make PLAT=lnx32e clean	- (インテル ® 64 アーキテクチャー)
make PLAT=lnx64 clean	- (IA-64 アーキテクチャー)

コンパイラー依存の関数と Fortran 90 モジュール

コンパイラーがそのランタイム・ライブラリー (RTL) で解決されるオブジェクト・コード関数呼び出しを行うと常に、コンパイラー依存の関数が使用されます。適切な RTL なしでこれらのコードをリンクすると、未定義シンボルになります。インテル ® MKL は、RTL の依存関係を最小限に抑えるように設計されています。

依存関係が発生する場合、サポートする RTL がインテル ® MKL とともに提供されます。インテル ® MKL クラスター・ソフトウェアに関連するものを除くと、このような RTL の唯一の例は、インテル ® コンパイラーでコンパイルされる OpenMP* コード用の libguide および libiomp です。libguide および libiomp は、インテル ® MKL でスレッド化されたコードをサポートしています。

RTL 依存関係が発生する可能性のあるほかの場合、関数はソースコードで提供されます。コンパイラーでコードをコンパイルするのはユーザーの責任です。

特に、Fortran 90 モジュールは、RTL をサポートするコンパイラー固有のコード生成が必要になるため、インテル ® MKL はこれらのモジュールをソースコードで提供しています。

混在言語プログラミングとインテル® MKL

付録 A は、各インテル® MKL 関数領域でサポートされているプログラミング言語の一覧です。しかし、インテル® MKL ルーチンを異なる言語環境から呼び出すこともできます。このセクションでは、混在言語プログラミングを使用してこの呼び出しを行う方法を説明します。

LAPACK、BLAS、および CBLAS ルーチンの C 言語環境からの呼び出し

すべてのインテル® MKL 関数領域で C と Fortran 環境の両方をサポートしているとは限りません。インテル® MKL Fortran 形式の関数を C/C++ 環境で使用するには、この後で説明されている LAPACK と BLAS の特定の規則に従う必要があります。

LAPACK

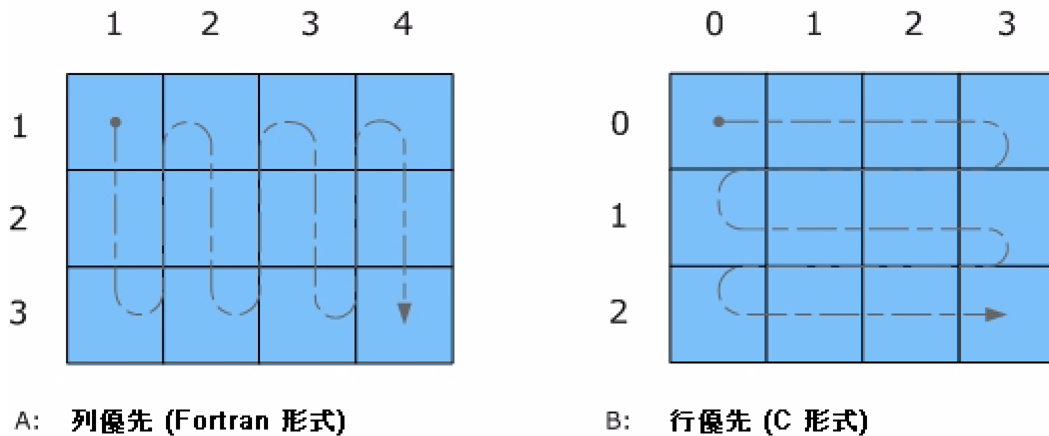
C 言語プログラムから呼び出す場合、LAPACK ルーチンは Fortran 形式です。以下の Fortran 形式呼び出し規則に従っていることを確認してください。

- 変数を値ではなくアドレスで渡します。

  例 7-1 および例 7-2 の関数呼び出しを参照してください
- データを Fortran 形式、つまり行優先ではなく列優先で格納します。

  C で採用されている行優先で、配列が格納されているメモリーを全検索すると、最後の配列インデックスが最も速く変更され、最初の配列インデックスが最も遅く変更されます。Fortran 形式の列優先では、最後のインデックスが最も遅く変更され、最初のインデックスが最も速く変更されます (図 7-1 の 2D 配列を参照)。

図 7-1 列優先と行優先



例えば、サイズ m x n の 2 次元行列 A が 1 次元配列 B に格納されている場合、行列の要素は以下のようにアクセスされます。

C の場合 : `A[i][j] = B[i*n+j]` (`i=0, ... , m-1, j=0, ... , n-1`)

Fortran の場合 : `A(i,j) = B(j*m+i)` (`i=1, ... , m, j=1, ... , n`)

LAPACK ルーチンを C から呼び出す場合、LAPACK ルーチンの名前には大文字と小文字の両方を使用できます (末尾の下線の有無を含む)。例えば、名前 dgetrf、DGETRF、dgetrf_、DGETRF_ は等価です。

BLAS

BLAS ルーチンは Fortran 形式のルーチンです。BLAS ルーチンを C 言語プログラムから呼び出す場合、Fortran 形式の呼び出し規則に従う必要があります。

- 変数を値ではなくアドレスで渡します。
- データを Fortran 形式、つまり行優先ではなく列優先で格納します。

これらの規約の詳細は、「LAPACK」セクションを参照してください。BLAS ルーチンを C から呼び出す方法は、「例 7-1」を参照してください。

BLAS ルーチンを C から呼び出す場合、BLAS ルーチンの名前には大文字と小文字の両方を使用できます (末尾の下線の有無を含む)。例えば、名前 dgemm、DGEMM、dgemm_、DGEMM_ は等価です。

CBLAS

BLAS ルーチンを C から呼び出す別の方法は、CBLAS インターフェイスを使用する方法です。

CBLAS は、BLAS ルーチンの C 形式のインターフェイスです。通常の C 形式の呼び出しを使用して CBLAS ルーチンを呼び出すことができます。CBLAS インターフェイスを使用している場合、ヘッダーファイル mkl.h によりすべての関数の列挙値とプロトタイプが指定されるため、プログラム開発が単純化されます。ヘッダーはプログラムが C++ コンパイラーでコンパイルされているかどうかを判断し、コンパイルされている場合、インクルード・ファイルは C++ コンパイル用に設定されます。例 7-3 は、CBLAS インターフェイスの使用例です。

C/C++ コードで複素数を返す BLAS 関数の呼び出し

C から複素数を返す BLAS 関数への呼び出しを制御している場合は注意が必要です。これらは Fortran 関数であり、複素数の戻り値の制御は C と Fortran で異なるため、問題が発生します。しかし、Fortran では、通常の関数呼び出しに加えて、関数が C プログラムから呼び出されたときに複素数の戻り値を返すように、関数をサブルーチンとして呼び出すことができます。Fortran 関数がサブルーチンとして呼び出された場合、戻り値は呼び出しシーケンスで最初のパラメーターになります。C プログラマーは、この違いに注意してください。

以下に呼び出し方法の違いを示します。

通常の Fortran 関数呼び出し : `result = cdotc( n, x, 1, y, 1 )`

関数をサブルーチンとして

呼び出す場合 : `call cdotc( result, n, x, 1, y, 1)`

C から関数を呼び出す場合

(隠しパラメーターが表示されている点に注意): `cdotc( &result, &n, x, &one, y, &one )`

注: インテル® MKL は、BLAS のサブルーチン名に大文字と小文字の両方を使用できます (末尾の下線の有無を含む)。このため、名前 cdotc、CDOTC、cdotc_、CDOTC_ はすべて使用できます。

上記の例を使用して、C および C++ から複素数を返すレベル 1 BLAS 関数を呼び出すことができます。しかし、CBLAS インターフェイスを使用するとより簡単です。例えば、以下のように CBLAS インターフェースを使用して、同じ関数を呼び出すことができます。

```
cblas_cdotu( n, x, 1, y, 1, &result )
```

注: この場合、明らかに複素数が返されます。

次の例は、C プログラムから BLAS レベル 1 の複素関数 zdotc() を呼び出す方法を表しています。この関数は、2 つの倍精度複素ベクトルのドット積を計算します。

この例では、複素数型のドット積が構造体 c に返されます。

例 7-1 複素 BLAS レベル 1 関数の C からの呼び出し

```
#include "mkl.h"
#define N 5
void main()
{
int n, inca = 1, incb = 1, i;
typedef struct{ double re; double im; } complex16;
complex16 a[N], b[N], c;
void zdotc();
n = N;
```

例 7-1　　　　複素 BLAS レベル 1 関数の C からの呼び出し (続き)

```
for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "複素ドット積 : ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

以下は C++ 実装です。

例 7-2　　　　複素 BLAS レベル 1 関数の C++ からの呼び出し

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;
extern "C" void zdotc (complex16*, int *, complex16 *, int *,
complex16 *, int *);

#define N 5

void main()
{
int n, inca = 1, incb = 1, i;

complex16 a[N], b[N], c;

n = N;

for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
zdotc(&c, &n, a, &inca, b, &incb );
printf( "複素ドット積 : ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

以下の実装は CBLAS を使用しています。

例 7-3 BLAS を C から直接呼び出す代わりに CBLAS インターフェイスを使用

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;

extern "C" void cblas_zdotc_sub ( const int , const complex16 *,
    const int , const complex16 *, const int, const complex16*);

#define N 5

void main()
{

int n, inca = 1, incb = 1, i;

complex16 a[N], b[N], c;
n = N;
for( i = 0; i < n; i++ ){

a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
cblas_zdotc_sub(n, a, inca, b, incb,&c );
printf( "複素ドット積 : ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

インテル® MKL 関数の Java アプリケーションからの呼び出し

このセクションでは、インテル® MKL パッケージで提供されるサンプルとライブラリー関数を Java から呼び出す方法を説明します。

インテル® MKL の Java サンプル

Java は、開発者である米 Sun Microsystems 社によって、WORA (Write Once Run Anywhere、一度プログラムを書けばどこでも実行できる) 言語と位置付けられています。インテル® MKL は、大部分のデスクトップ PC とノートブック PC、多くのワークステーションとサーバーをカバーする、広範囲なオペレーティング・システムとプロセッサー用にさまざまなエディションを提供することで、WORA 哲学を部分的にサポートし、Java アプリケーションの高速化を支援します。

インテル® MKL には以下のディレクトリーにさまざまな Java のサンプルが含まれています。

<*mkl ディレクトリー*>/examples/java

以下のインテル® MKL 関数用のサンプルが提供されています。

- CBLAS の ?gemm、?gemv、および ?dot ファミリー
- 非クラスター FFT 関数の完全なセット
- 1 次元の畳み込み / 相関用 ESSL[1] 形式の関数
- VSL 乱数生成器 (RNG)、ユーザー定義のものとファイル・サブルーチンを除く
- GetErrorCallBack、SetErrorCallBack、および ClearErrorCallBack を除く VML 関数

サンプルのソースは以下のディレクトリーにあります。

<*mkl ディレクトリー*>/examples/java/examples

サンプルは Java で記述されています。サンプルでは、以下の種類のデータを使用しています。

- 1 次元および 2 次元データシーケンス
- データの実数型と複素数型
- 単精度と倍精度

ただし、サンプルで使用されているラッパーは以下のことを行いません。

- 巨大な配列 (要素が 2 億以上) の使用
- ネイティブメモリー中の配列の処理
- 関数パラメーターの正当性の確認
- パフォーマンスの最適化

インテル® MKL とバインドするため、サンプルは JNI (Java Native Interface) を使用しています。JNI のドキュメントは、
http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html (英語) を参照してください。

Java のサンプルには、バインドを行う JNI ラッパーも含まれています。ラッパーはサンプルに依存しません。各自の Java アプリケーションで使用することもできます。CBLAS、FFT、VML、VSL RNG、および ESSL 形式の畳み込み / 相関関数のラッパーは互いに依存しません。

ラッパーをビルドするには、サンプルを実行してください (詳細は、「サンプルの実行」セクションを参照)。メイクファイルを実行すると、ラッパーのバイナリーとサンプルがビルドされます。サンプルを実行すると、
<*mkl ディレクトリー*>/examples/java に以下のディレクトリーが作成されます。

- docs
- include
- classes
- bin
- _results

1. IBM ESSL* ライブラリー

docs、include、classes、および bin ディレクトリーには、ラッパーのバイナリーとドキュメントが含まれます。_results ディレクトリーには、テスト結果が含まれます。

Java プログラマーにとっては、ラッパーは次のような Java クラスに見えます。

- com.intel.mkl.CBLAS
- com.intel.mkl.DFTI
- com.intel.mkl.ESSL
- com.intel.mkl.VML
- com.intel.mkl.VSL

特定のラッパーとサンプルのクラスのドキュメントは、サンプルのビルドおよび実行中に Java ソースから生成されます。ドキュメントは、ビルドスクリプトを実行すると docs ディレクトリーに作成される、以下のファイルから参照することができます。

<mkl ディレクトリー>/examples/java/docs/index.html

CBLAS、VML、VSL RNG、および FFT 用の Java ラッパーは、基本となるネイティブ関数に直接対応するインターフェイスを確立します。機能とパラメーターは、『インテル® MKL リファレンス・マニュアル』を参照してください。ESSL 形式の関数用のインターフェイスは、com.intel.mkl.ESSL クラス用に生成されたドキュメントで説明されています。

各ラッパーは、Java のインターフェイス部分と C で記述された JNI スタブで構成されています。ソースは以下のディレクトリーにあります。

<mkl ディレクトリー>/examples/java/wrappers

CBLAS と VML 用のラッパーの Java と C 部分はどちらも標準的なアプローチを採用しているため、不足している CBLAS 関数をカバーするために使用できます。

FFT 用のラッパーは、FFT ディスクリプター・オブジェクトのライフサイクルをサポートしているため、より複雑です。単一フーリエ変換を計算するには、アプリケーションはネイティブ FFT ディスクリプターの同じコピーを使用して FFT ソフトウェアを複数回呼び出す必要があります。ラッパーは、仮想マシンが Java バイトコードを実行する間、ネイティブ・ディスクリプターを保持するハンドラークラスを提供します。

VSL RNG 用のラッパーは、FFT 用のラッパーと似ています。ラッパーは、ストリームステートのネイティブ・ディスクリプターを保持するハンドラークラスを提供します。

畳み込み / 相関関数用のラッパーは、" タスク・ディスクリプター " と同様のライフサイクルを仮定し、VSL インターフェイスの難易度を緩和します。ラッパーは、1 次元でより単純な、これらの関数の ESSL 形式の関数を使用します。JNI スタブは、C で記述された ESSL 形式のラッパーにインテル® MKL 関数をラップし、ネイティブメソッドへの単一呼び出しにタスク・ディスクリプターのライフサイクルを " パック " します。

ラッパーは JNI 仕様 1.1 および 5.0 を満たしているため、新しい Java のすべての実装で動作します。

サンプルとラッパーの Java 部分は、「*The Java Language Specification (First Edition)*」で説明されている Java 言語用に記述され、1990 年代後半に登場したインナークラスの機能が拡張されています。この言語バージョンのレベルは、Sun の Java SDK (ソフトウェア開発キット) のすべてのバージョンと、バージョン 1.1.5 以降の互換性のある実装をサポートしています。

C 言語レベルは、インテル ® MKL インターフェイスと JNI ヘッダーファイルで必要な、整数と浮動小数点データ型に関する追加の仮定を含む " 標準 C" (C89) です。つまり、ネイティブ `float` および `double` データ型は、JNI `jfloat` および `jdouble` データ型とそれぞれ同じである必要があります。また、ネイティブ `int` データ型は、4 バイト長である必要があります。

サンプルの実行

Java のサンプルは、インテル ® MKL でサポートされている C/C++ コンパイラーをすべてサポートしています。メイクファイルを実行するには、Linux* オペレーティング・システムで提供される make ユーティリティーが必要です。

Java のサンプルを実行するには、Java コードのコンパイルと実行に Java SDK が必要です。Java 実装はコンピューターにインストールされているか、ネットワーク経由で利用可能である必要があります。SDK は、各ベンダーの Web サイトからダウンロードできます。

サンプルは、Java 2 SE SDK のすべてのバージョンで動作するように作成されていますが、以下の Java 実装でのみテストされています。

- Sun Microsystems 社 (http://sun.com)
- BEA (http://bea.com)

サポートしている Java SDK のバージョンについては、『インテル ® MKL リリースノート』を参照してください。

注 : Sun Microsystems 社の実装は、IA-32 およびインテル ® 64 アーキテクチャー対応のプロセッサーのみをサポートしています。BEA の実装は、インテル ® Itanium® 2 プロセッサーもサポートしています。

コンピューターに JRE (Java ランタイム環境) がインストールされている場合でも、以下のツールをサポートする JDK が必要です。

- java
- javac
- javah
- javadoc

これらのツールをサンプルのメイクファイルで利用できるようにするには、以下のように、`JAVA_HOME` 環境変数を設定し、`PATH` 環境変数に JDK バイナリーのディレクトリーを追加する必要があります。

```
export JAVA_HOME=/home/<user name>/jdk1.5.0_09
```

```
export PATH=${JAVA_HOME}/bin:${PATH}
```

`JDK_HOME` 環境変数に値が割り当てられている場合は、この環境変数をクリアする必要もあります。

```
unset JDK_HOME
```

サンプルを開始するには、インテル ® MKL の Java サンプルが含まれているディレクトリーにあるメイクファイルを使用します。

```
make {so32|soem64t|so64} [function=…] [compiler=…]
```

ターゲット (`so32` など) を指定しないで開始した場合、メイクファイルは、ターゲット、*`function`*、*`compiler`* パラメーターについてのヘルプを表示します。

サンプルのリストは、同じディレクトリーにある `examples.lst` ファイルを参照してください。

既知の制限事項

3 種類の制限があります。

- 機能
- パフォーマンス
- 既知の問題

機能 : インテル® MKL の Java サンプルのように、ラッパーを使用して Java 環境から呼び出された場合、一部のインテル® MKL 関数が正常に動作しない可能性があります。「インテル® MKL の Java サンプル」セクションにリストされている、これらの特定の CBLAS、FFT、VML、VSL RNG、および畳み込み / 相関関数は、Java 環境でテストされています。このため、各自の Java アプリケーションでは、これらの CBLAS、FFT、VML、VSL RNG、および畳み込み / 相関関数用の Java ラッパーを使用してください。

パフォーマンス : インテル® MKL 関数はピュア Java で記述された同様の関数よりも高速です。しかし、これらのラッパーではパフォーマンスが主な目標ではないことに注意してください。目標は、コードのサンプルを提供することです。このため、Java アプリケーションから呼び出されたインテル® MKL 関数は、C/C++ または Fortran で記述されたプログラムから呼び出された同じ関数よりも遅くなります。

既知の問題 : インテル® MKL には (リリースノートに示されている) 既知の問題があります。また、Java SDK の異なるバージョンでは互換性がないものがあります。サンプルおよびラッパーには、サンプルを動作させるための、これらの問題の回避策が用意されています。サンプルおよびラッパーのソースコードに記述されている、回避策についてのコメントを参照してください。

コーディングのヒント

8

本章は、インテル ® マス・カーネル・ライブラリー (インテル ® MKL) を使用したプログラミングについて説明するもう 1 つの章です。第 7 章では、一般的な言語固有のプログラミング・オプションを説明しましたが、本章では特定の用途を満たすために役立つコーディングのヒントを示します。本章では数値計算安定性を達成する方法についてのヒントを紹介します。パフォーマンスとメモリー管理に関連するコーディングのヒントは、第 6 章を参照してください。

数値計算安定性のためのデータの整列

線形代数ルーチン (LAPACK, BLAS) がビットごとに同一の入力に適用され、配列のアライメントが異なるか、または、計算が異なるプラットフォーム上や異なるスレッド数で行われる場合、出力はビットごとに同一ではないため、適切な誤差範囲に収まらない場合があります。バージョンによってルーチンの実装が異なる場合があるため、インテル ® MKL のバージョンは出力の数値計算安定性に影響します。指定されたインテル ® MKL バージョンで、以下の条件がすべて満たされる場合、出力はビットごとに同一になります。

- 出力が同じプラットフォームで得られる
- 入力がビットごとに同一である
- 入力配列が 16 バイト境界で同一にアライメントされている

ユーザーの管理下にある最初の 2 つの条件とは異なり、配列はデフォルトではアライメントされていません。例えば、`malloc` を使用して動的に割り当てられた配列は、16 バイトではなく 8 バイト境界でアライメントされます。数値的に安定した出力が必要な場合、正しくアライメントされたアドレスを得るため、以下のようなコードを使用してください。

例 8-1　16 バイト境界でアドレスをアライメント

```
// C 言語
...
#include <stdlib.h>
...
void *ptr, *ptr_aligned;
ptr = malloc( sizeof(double)*workspace + 16 );
ptr_aligned = ( (size_t)ptr%16 ?(void*)
((size_t)ptr+16-((size_t)ptr%16)) : ptr );
// MKL を使用してプログラムを呼び出し、16 ビット境界でアライメントされたアドレス
を渡します。
mkl_app( ptr_aligned );
...
free( ptr );


! Fortran 言語
...
double precision darray( workspace+1 )
! MKL を使用してプログラムを呼び出し、16 ビット境界でアライメントされたアドレスを
渡します。
if( mod( %loc(darray), 16 ).eq.0 ) then
     call mkl_app( darray(1) )
else
     call mkl_app( darray(2) )
end if
```

インテル® マス・カーネル・ライブラリー・クラスター・ソフトウェアの使用

9

本章は、インテル® MKL ScaLAPACK とクラスター FFT の使用法について、C と Fortran 固有のリンク例を含む、関数領域を使用するアプリケーションのリンク方法を主に説明します。また、サポートしている MPI の情報も示します。

詳細なインテル® MKL ディレクトリー構造は、第 3 章の表 3-7 を参照してください。

利用可能なドキュメントおよび `doc` ディレクトリーについての情報は、同じ章の表 3-8 を参照してください。

クラスター用の MP LINPACK ベンチマークについての情報は、第 10 章の「Intel® Optimized MP LINPACK Benchmark for Clusters」セクションを参照してください。

インテル® MKL ScaLAPACK および FFT は、MPICH-1.2.x およびインテル® MPI をサポートしています。

ScaLAPACK を呼び出すプログラムをリンクするには、まず MPI アプリケーションのリンク方法を知っておく必要があります。

通常は、正しい MPI ヘッダーファイルを使用する *mpi* スクリプト *mpicc* または *mpif77* (C または Fortran77 スクリプト) の使用も含まれます。例えば、`/opt/mpich` にインストールした MPICH を使用している場合、通常は `/opt/mpich/bin/mpicc` および `/opt/mpich/bin/mpif77` がコンパイラー・スクリプト、`/opt/mpich/lib/libmpich.a` がインストールに使用したライブラリーです。

ScaLAPACK および クラスター FFT とのリンク

インテル® MKL ScaLAPACK とクラスター FFT の両方またはいずれか一方とリンクするには、以下の一般的な形式を使用します。

```
<<MPI> リンカースクリプト > < リンクするファイル >            \
    -L< クラスター MKL パス > < クラスター MKL ライブラリー > \
    <BLACS> <MKL コア・ライブラリー >                        \
```

説明

<MPI> は、MPI 実装 (MPICH、インテル® MPI 1.x、インテル® MPI 2.x、インテル® MPI 3.x) のいずれか 1 つです。

<BLACS> は、`-lmkl_blacs`、`-lmkl_blacs_intelmpi`、`-lmkl_blacs_intelmpi20`、`-mkl_blacs_openmpi` のいずれか 1 つです。

<MKL クラスター・ライブラリー> は、`-lmkl_scalapack-core` と `-lmkl_cdft_core` の両方またはいずれか一方です。

<MKL コア・ライブラリー> は、ScaLAPACK の場合は *<MKL LAPACK & MKL カーネル・ライブラリー>*、クラスター FFT の場合は *<MKL カーネル・ライブラリー>* です。

<MKL LAPACK & カーネル・ライブラリー> は、第 5 章の「リンクコマンドの構文」セクションの最初に記述されているように、スレッド化をサポートするためにリンクされている、LAPACK、プロセッサー最適化カーネル、スレッド化ライブラリー、およびシステム・ライブラリーです。

<<MPI> リンカースクリプト> および *<BLACS>* ライブラリーは MPI のバージョンに対応する必要があることに注意してください。例えば、インテル® MPI 2.x の場合、*<インテル® MPI 2.x リンカースクリプト>* および `libmkl_blacs_intelmpi20` ライブラリーを使用します。インテル® MPI 3.0 の場合、`libmkl_blacs_intelmpi20` も使用する必要があります。

インテル® MKL ライブラリーのリンクについては、第 5 章「アプリケーションと インテル® マス・カーネル・ライブラリーのリンク」を参照してください。

スレッド数の設定

OpenMP* ソフトウェアは、`OMP_NUM_THREADS` 環境変数を使用します。インテル® MKL 10.0 では、`MKL_NUM_THREADS` や `MKL_DOMAIN_NUM_THREADS` のような、スレッド数を設定するほかの方法が追加されました (第 6 章の「「新しいスレッド化コントロールの使用」」セクションを参照)。適切な環境変数がすべてのノードにおいて同じで正しい値になっていることを確認してください。また、インテル® MKL 10.0 では、デフォルトのスレッド数が 1 はでなくなりました。デフォルトのスレッド数はコンパイラーに応じて設定されます。インテル® コンパイラー・ベースのスレッド化レイヤー (`libmkl_intel_thread.a`) では、この値は OS の CPU の数です。例えば、ノードあたりの MPI ランクの数とノードあたりのスレッド数の両方が 1 よりも大きい場合、スレッド数が過剰に指定されないように注意してください。

`OMP_NUM_THREADS` 環境変数を設定する最良の方法は、ログイン環境で設定することです。`mpirun` がすべてのノードでデフォルトシェルを開始することに注意してください。このため、ヘッドノードでこの値を変更してから実行 (SMP システムで動作) しても、各自のプログラムに関する限り、変数の変更は有効になりません。 `.bashrc` では、以下のように先頭に行を追加することができます。

```
OMP_NUM_THREADS=1; export OMP_NUM_THREADS
```

MPICH を使用してノードごとに複数の CPU を実行することは可能ですが、そのためには MPICH を構築する必要があります。特定の MPICH アプリケーションはスレッド化環境では正常に動作しない場合があることに注意してください (*『リリースノート』* の「既知の制限事項」セクションを参照)。複数の CPU を使用する最も安全な方法は、最も高速ではありませんが、`OMP_NUM_THREADS=1` に設定してプロセッサーごとに 1 つの MPI プロセスを実行する方法です。`OMP_NUM_THREADS=1` の組み合わせが正常に動作することを常に確認してください。

共有ライブラリーの使用

すべての必要な共有ライブラリーはランタイムにすべてのノードで見えている必要があります。この状況を実現する 1 つの方法は、 `.bashrc` ファイルで `LD_LIBRARY_PATH` 環境変数を使用してこれらのライブラリーを指すことです。インテル® MKL が 1 つのノードにのみインストールされている場合、インテル® MKL アプリケーションをビルドするとき静的にリンクしてください。

インテル ® コンパイラーまたは GNU コンパイラーは、インテル ® MKL を使用するプログラムをコンパイルすることができます。しかし、MPI 実装とコンパイラーが正しく一致していることを確認してください。

ScaLAPACK テスト

IA-32、IA-64、またはインテル ® 64 アーキテクチャーで NetLib ScaLAPACK テストをビルドするには、リンクコマンドに libmkl_scalapack_core.a を追加します。

ScaLAPACK およびクラスター FFT とのリンクの例

クラスター・ライブラリーのアーキテクチャー固有のディレクトリーの詳細な構造は、第 3 章の「詳細なディレクトリー構造」セクションを参照してください。

C モジュールの例

以下の条件が満たされていると仮定します。

- MPICH 1.2.5 以降が /opt/mpich にインストールされている。
- インテル ® MKL 10.0 が /opt/intel/mkl/10.0.xxx (xxx はインテル ® MKL パッケージ番号。例 :/opt/intel/mkl/10.0.039) にインストールされている。
- インテル ® C コンパイラー 8.1 以降を使用していて、メインモジュールが C である。

IA-32 アーキテクチャー・ベースのシステムのクラスターで ScaLAPACK をリンクするには: 以下のコマンドを使用します。

```
/opt/mpich/bin/mpicc < リンクするユーザーファイル >     \
     -L/opt/intel/mkl/10.0.xxx/lib/32           \
     -lmkl_scalapack_core                       \
     -lmkl_blacs                                \
     -lmkl_lapack                               \
     -lmkl_intel -lmkl_intel_thread -lmkl_core \
     -lguide                                    \
     -lpthread
```

IA-64 アーキテクチャー・ベースのシステムのクラスターでクラスター FFT をリンクするには: 以下のコマンドを使用します。

```
/opt/mpich/bin/mpicc < リンクするユーザーファイル >     \
     -L/opt/intel/mkl/10.0.xxx/lib/64           \
     -lmkl_cdft_core                            \
     -lmkl_blacs_ilp64                          \
     -lmkl_intel -lmkl_intel_thread -lmkl_core \
     -lguide -lpthread
```

Fortran モジュールの例

以下の条件が満たされていると仮定します。

- インテル® MPI 3.0 が `/opt/intel/mpi/3.0` にインストールされている。
- インテル® MKL 10.0 が `/opt/intel/mkl/10.0.xxx` (`xxx` はインテル® MKL パッケージ番号。例 :`/opt/intel/mkl/10.0.039`) にインストールされている。
- インテル® Fortran コンパイラー 8.1 以降を使用していて、メインモジュールが Fortran である。

IA-64 アーキテクチャー・ベースのシステムのクラスターで ScaLAPACK をリンクするには、: 以下のコマンドを使用します。

```
/opt/intel/mpi/3.0/bin/mpiifort < リンクするユーザーファイル >        \
     -L/opt/intel/mkl/10.0.xxx/lib/64                            \
     -lmkl_scalapack_lp64                                        \
     -lmkl_blacs_intelmpi20_lp64                                 \
     -lmkl_lapack                                                \
     -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core              \
     -lguide                                                     \
     -lpthread
```

IA-64 アーキテクチャー・ベースのシステムのクラスターでクラスター FFT をリンクするには、: 以下のコマンドを使用します。

```
/opt/intel_mpi_10/bin/mpiifort < リンクするユーザーファイル > \
     -L/opt/intel/mkl/10.0.xxx/lib/64                  \
     -lmkl_cdft_core                                   \
     -lmkl_blacs_intelmpi_ilp64                        \
     -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core    \
     -lguide -lpthread
```

ScaLAPACK とリンクされたバイナリーは、ほかの MPI アプリケーションと同じ方法で動作します (詳細は、MPI 実装に含まれているドキュメントを参照してください)。例えば、スクリプト `mpirun` は MPICH 1.2.x および OpenMPI の場合に使用され、MPI プロセスの数は `-np` で設定されます。MPICH 2.0 およびすべてのインテル® MPI の場合、アプリケーションを実行する前にデーモンを開始する必要があります。実行にはスクリプト `mpiexec` を使用します。

他のリンク例は、インテル® MKL サポート Web サイト http://www.intel.com/support/performancetools/libraries/mkl/ (英語) を参照してください。

LINPACK ベンチマークと MP LINPACK ベンチマーク

10

本章は、Intel® Optimized LINPACK Benchmark for Linux* と Intel® Optimized MP LINPACK Benchmark for Clusters について説明します。

Intel® Optimized LINPACK Benchmark for Linux

Intel® Optimized LINPACK Benchmark は、LINPACK 1000 ベンチマークを一般化したものです。このベンチマークは、稠密な (`real*8`) 連立線形方程式 (*Ax=b*) を解き、因数分解して解くためにかかった時間を測定し、時間をパフォーマンス比率に変換して、結果の精度をテストします。一般化により、1000 を超える方程式 (*N*) を解くことができます。結果の精度を保証するため部分的なピボット演算を使用します。

このベンチマークは、コンパイルされたコードのみを対象とするベンチマークであるため、LINPACK 100 のパフォーマンスを報告するためには使用しないでください。このベンチマークは、単一プラットフォームで実行する共有メモリー (SMP) 実装です。同じベンチマークの分散型メモリーバージョンである MP LINPACK と混同しないようにしてください。また、このベンチマークを、LAPACK ライブラリーで拡張された LINPACK ライブラリーと混同しないでください。

インテルは、HPL を使用するよりも簡単にインテル ® プロセッサー・ベースのシステムで高い LINPACK ベンチマーク結果が得られる LINPACK ベンチマークの最適化バージョンを提供しています。SMP マシンのベンチマークには、このパッケージを使用してください。

このソフトウェアの詳細は、http://developer.intel.com/software/products/ を参照してください。

内容

Intel® Optimized LINPACK Benchmark for Linux には、以下のファイルが含まれています。ファイルは、インテル ® MKL ディレクトリーの `/benchmarks/linpack/` サブディレクトリーにあります (「表 3-1」を参照)。

表 10-1　LINPACK Benchmark の内容

./benchmarks/linpack/	
`linpack_itanium`	インテル ® Itanium® 2 プロセッサー・ベースのシステム用 64 ビット・プログラム

表 10-1 LINPACK Benchmark の内容 (続き)

./benchmarks/linpack/	
linpack_xeon32	ストリーミング SIMD 拡張命令 3 (SSE3) 対応 / 非対応インテル® Xeon® プロセッサーまたはインテル® Xeon® プロセッサー MP ベースのシステム用 32 ビット・プログラム
linpack_xeon64	インテル® 64 アーキテクチャー対応インテル® Xeon® プロセッサー・ベースのシステム用 64 ビット・プログラム
runme_itanium	linpack_itanium 用に事前に定義された問題セットを実行するためのサンプル・シェル・スクリプト。OMP_NUM_THREADS は 8 プロセッサーに設定されます。
runme_xeon32	linpack_xeon32 用に事前に定義された問題セットを実行するためのサンプル・シェル・スクリプト。OMP_NUM_THREADS は 2 プロセッサーに設定されます。
runme_xeon64	linpack_xeon64 用に事前に定義された問題セットを実行するためのサンプル・シェル・スクリプト。OMP_NUM_THREADS は 4 プロセッサーに設定されます。
lininput_itanium	runme_itanium スクリプト用に事前に定義された問題の入力ファイル。
lininput_xeon32	runme_xeon32 スクリプト用に事前に定義された問題の入力ファイル。
lininput_xeon64	runme_xeon64 スクリプト用に事前に定義された問題の入力ファイル。
lin_itanium.txt	runme_itanium スクリプトを実行した結果。
lin_xeon32.txt	runme_xeon32 スクリプトを実行した結果。
lin_xeon64.txt	runme_xeon64 スクリプトを実行した結果。
help.lpk	標準ヘルプファイル。
xhelp.lpk	拡張ヘルプファイル。

ソフトウェアの実行

指定したシステムで事前に定義されたサンプル問題サイズの結果を得るには、次のいずれかのコマンドを入力します。

```
./runme_itanium
./runme_xeon32
./runme_xeon64
```

ほかの問題サイズでソフトウェアを実行する方法は、プログラムに含まれている拡張ヘルプを参照してください。拡張ヘルプは、以下のように "-e" オプションを指定してプログラムを実行すると表示されます。

```
./xlinpack_itanium -e
./xlinpack_xeon32 -e
./xlinpack_xeon64 -e
```

データ入力ファイル lininput_itanium、lininput_xeon32、および lininput_xeon64 は、単なる例として提供されています。プロセッサー数やメモリー量が異なるシステムでは入力ファイルを変更する必要があります。入力ファイルを変更する適切な方法は、拡張ヘルプを参照してください。

各入力ファイルでは、少なくとも以下の量のメモリーが必要です。

lininput_itanium	16GB
lininput_xeon32	2GB
lininput_xeon64	16GB

システムのメモリー量が上記のデータ入力ファイルで必要なメモリー量よりも少ない場合、拡張ヘルプの指示に従って既存のデータ入力ファイルを編集するか、新しいデータ入力ファイルを作成してください。

各サンプルスクリプトでは、OMP_NUM_THREADS 環境変数を使用してターゲットのプロセッサー数を設定します。異なる物理プロセッサー数でパフォーマンスを最適化するには、該当する行を適切な値に変更してください。スレッド数を設定しないで Intel® Optimized LINPACK Benchmark を実行すると、OS に従ってデフォルトのコア数が設定されます。この環境変数の設定は、runme_* サンプルスクリプトで行われています。設定が使用している環境と一致しない場合、スクリプトを編集してください。

既知の制限事項

Intel Optimized LINPACK Benchmark for Linux には、以下の既知の制限があります。

- Intel Optimized LINPACK Benchmark は、複数のプロセッサーを使用して効率的にスレッド化されます。このため、ハイパースレッディング・テクノロジー対応のマルチプロセッサー・システムで最適なパフォーマンスを得るには、オペレーティング・システムが物理プロセッサーにスレッドを割り当てるように、ハイパースレッディング・テクノロジーを無効にしてください。
- 不完全なデータ入力ファイルが指定されると、バイナリーはハングアップするか失敗します。正しいデータ入力ファイルの作成方法は、データ入力ファイルのサンプルまたは拡張ヘルプを参照してください。

Intel® Optimized MP LINPACK Benchmark for Clusters

Intel® Optimized MP LINPACK Benchmark for Clusters は、テネシー大学ノックスビル校 (UTK) の Innovative Computing Laboratories (ICL) が提供している HPL 1.0a をベースに修正、追加したものです。ベンチマークは、Top 500 (http://www.top500.org を参照) の実行に使用することができます。ベンチマークを使用するには、HPL ディストリビューションと使用法について熟知している必要があります。このパッケージは、HPL をより便利に使用できるように、追加の拡張とバグフィックスが行われています。benchmarks/mp_linpack ディレクトリーには、長時間の実行における検索時間を最小限に抑えるための手法が加えられています。

Intel® Optimized MP LINPACK Benchmark for Clusters は、LINPACK の超並列対応版である Massively Parallel MP LINPACK ベンチマークの実装です。HPL コードは基礎として使用されています。このベンチマークは、ランダムで稠密な (real*8) 連立線形方程式 (*Ax=b*) を解き、因数分解して解くため

にかかった時間を測定し、時間をパフォーマンス比率に変換して、結果の精度をテストします。メモリーに収まる任意のサイズ (*N*) の連立方程式を解くことができます。ベンチマークは、結果の精度を保証するために完全な行ピボット演算を使用します。

このベンチマークは、共有メモリーマシンの LINPACK パフォーマンスを報告するために使用しないでください。その場合は、代わりに Intel® Optimized LINPACK Benchmark を使用してください。このベンチマークは、分散型メモリーマシンで使用するものです。

インテルは、HPL を使用するよりも簡単にインテル® プロセッサー・ベースのシステムで高い LINPACK ベンチマーク結果が得られる LINPACK ベンチマークの最適化バージョンを提供しています。クラスターのベンチマークには、このパッケージを使用してください。用意されているバイナリーを使用するには、クラスターにインテル® MPI 3.x がインストールされている必要があります。インテル® MPI のランタイムバージョンは www.intel.com/software/products/cluster からダウンロードできます。

注 : MPI の異なるバージョンを使用する場合は、提供されている MP LINPACK ソースを使用してください。

パッケージには、テネシー大学ノックスビル校の Innovative Computing Laboratories (ICL) で開発されたソフトウェアが含まれていますが、これはテネシー大学や ICL が本製品を推奨あるいは販促していることを意味するものではありません。HPL 1.0a は特定の条件の下で再配布することができますが、このパッケージはインテル® MKL の使用許諾契約書に従います。

内容

Intel® Optimized MP LINPACK Benchmark for Clusters には、HPL 1.0a ディストリビューションとその修正が含まれています。ファイルの一覧は 表 10-2 を参照してください。ファイルは、インテル® MKL ディレクトリーの `/benchmarks/mp_linpack/` サブディレクトリーにあります (「表 3-1」を参照)。

表 10-2 MP LINPACK Benchmark の内容

./benchmarks/mp_linpack/	
`testing/ptest/HPL_pdtest.c`	HPL 1.0a コードに `ASYOUGO2_DISPLAY` (詳細は、「新機能」セクションを参照) で `DGEMM` 情報がキャプチャーされた場合に情報を表示する修正を加えたもの。
`src/blas/HPL_dgemm.c`	HPL 1.0a コードに `ASYOUGO2_DISPLAY` で指定された場合に `DGEMM` 情報をキャプチャーする修正を加えたもの。
`src/grid/HPL_grid_init.c`	HPL 1.0a コードに HPL 1.0 にない追加のグリッド試験を行う修正を加えたもの。
`src/pgesv/HPL_pdgesvK2.c`	HPL 1.0a コードに `ASYOUGO` および `ENDEARLY` の修正を加えたもの。

表 10-2　MP LINPACK Benchmark の内容

./benchmarks/mp_linpack/	
include/hpl_misc.h と hpl_pgesv.h	64 ビット・アドレス計算の許容が追加されたバグフィックス。
src/pgesv/HPL_pdgesv0.c	HPL 1.0a コードに ASYOUGO、ASYOUGO2、および ENDEARLY の修正を加えたもの。
testing/ptest/HPL.dat	HPL 1.0a のサンプル HPL.dat を修正したもの。
Make.ia32	(新規) IA-32 アーキテクチャー対応プロセッサー・ベースの Linux システム用のサンプル・アーキテクチャー make。
Make.em64t	(新規) インテル ® 64 アーキテクチャー対応プロセッサー・ベースの Linux システム用のサンプル・アーキテクチャー make。
Make.ipf	(新規) IA-64 アーキテクチャー対応プロセッサー・ベースの Linux システム用のサンプル・アーキテクチャー make。
次の 3 つのファイルは、単純なパフォーマンス・テストに利用できる、事前に構築された実行可能ファイルです。	
bin_intel/ia32/xhpl_ia32	(新規) IA-32 アーキテクチャーの Linux システム、インテル ® MPI 3.0 用の事前に構築されたバイナリー。
bin_intel/em64t/xhpl_em64t	(新規) インテル ® 64 アーキテクチャーの Linux システム、インテル ® MPI 3.0 用の事前に構築されたバイナリー。
bin_intel/ipf/xhpl_ipf	(新規) IA-64 アーキテクチャーの Linux システム、インテル ® MPI 3.0 用の事前に構築されたバイナリー。
HPL.dat	testing/ptest/HPL.dat のコピー。
nodeperf.c	(新規) クラスターの DGEMM 速度をテストするサンプル・ユーティリティー。

MP LINPACK の構築

サンプル・アーキテクチャー make がいくつか用意されています。使用している構成に合わせて、これらのファイルを以下のように編集することを推奨します。

- TOPdir を MP LINPACK が含まれているディレクトリーに設定します。
- MPI 変数、MPdir、MPinc、および MPlib を設定します。
- インテル ® MKL と使用するファイルの場所を指定します (LAdir、LAinc、LAlib)。
- コンパイラーおよびコンパイラー / リンカーオプションを調整します。

インテル ® 64 アーキテクチャー・ベースの Linux システムのような一部のサンプルケースでは、make には一般的な値が含まれています。しかし、HPL の構築についてよく理解した上で、これらの変数に適切な値を設定するようにしてください。

新機能

ツールセットは HPL 1.0a ディストリビューションと基本的に同一です。いくつかの変更は、オプションで指定してコンパイルしない限り無効です。以下の新機能があります。

ASYOUGO: 実行が進行するとともに、非侵入型のパフォーマンス情報を提供します。出力はわずかで、この情報はパフォーマンスに影響しません。情報を提供しなくても、多くの実行が長時間実行されるため、特に役立つ機能です。

ASYOUGO2: すべての DGEMM を傍受するため、わずかに侵入する付加的なパフォーマンス情報を提供します。

ASYOUGO2_DISPLAY: 実行内部の有効なすべての DGEMM のパフォーマンスを表示します。

ENDEARLY: いくつかのパフォーマンスのヒントを表示し、実行を早く終了します。

FASTSWAP: HPL のコードに LAPACK で最適化された DLASWP を挿入します。インテル ® Itanium® 2 プロセッサーで役立ちます。この機能を使用して試験することで最良の結果を決定できます。

クラスターのベンチマーク

クラスターのベンチマークを行うには、以下の手順に従ってください。ステップ 3. と 4. の繰り返しには特に注意してください。クラスターが達する最高のパフォーマンスを表す HPL パラメーター (HPL.dat で指定) を検索するため必要以上に繰り返されることがあります。

1. HPL をすべてのノードにインストールして有効にします。
2. ディストリビューションに含まれている nodeperf.c を実行し、すべてのノードで DGEMM のパフォーマンスを確認します。

   MPI およびインテル ® MKL を使用して nodeperf.c をコンパイルします。

   例 :

   ```
   mpicc -O3 nodeperf.c
   /opt/intel/mkl/10.0.xxx/lib/em64t/libmkl_em64t.a
   /opt/intel/mkl/10.0.xxx/lib/em64t/libguide.a -lpthread -o nodeperf
   ```

   (xxx はインテル ® MKL パッケージ番号)

   すべてのノードで nodeperf.c を起動することは、非常に大規模なクラスターでは特に有用です。実際に、ほかよりも 5% 遅く実行される特定のノード (例えば、738) にジョブが残ることがあります。MP LINPACK は、最も遅いノードに合わせて実行します。この場合、nodeperf を使用すると、悪いノードを見つけるためにクラスターで多くの小さな MP LINPACK 実行を行うことなく、潜在的な問題のスポットを素早く識別することができます。HPL を何度か実行した後、ゾンビプロセスが存在し、nodeperf が遅いノードを検索することは一般的です。検索はすべてのノードに対して一つずつ行われ、DGEMM のパフォーマンスに続けてホスト識別子が報告されます。このため、最後から 2 つめの数が大きいほど、ノードの実行が高速であったことになります。
3. 使用するクラスターに合わせて HPL.dat を編集します。

   詳細は、HPL のドキュメントを参照してください。ただし、少なくとも 4 つのノードで試すようにしてください。

4. ASYOUGO、ASYOUGO2 または ENDEARLY コンパイラー・オプションを使用して HPL を実行します (これらのオプションを使用することで、HPL が考察を行うよりも早くパフォーマンスに対する考察を行うことができます)。

   実行するときは、以下の推奨事項に従ってください。

   - 検索時間を短縮するため、HPL の MP LINPACK パッチ済みバージョンを使用してください。

     HPL のパッチ済みバージョンを使用することでパフォーマンスに影響を与えないようにする必要があります。このため、パフォーマンスに影響を与える可能性のある機能は、MP LINPACK では (この後で説明するように) コンパイラー・オプションとして提供されています 「検索時間を短縮するためのオプション」セクションで説明されている新しいオプションを使用しない場合、これらの変更は無効になります。拡張の主目的は、検索ソリューションを補助することです。

     HPL では、多くの異なるパラメーターの検索に長い時間がかかります。MP LINPACK では、最適な数を得ることが目標です。

     入力が修正されない場合、大きなパラメーター空間を検索する必要があります。実際、あらゆる入力の全数検索は、強力なクラスターでもかなりの時間がかかります。

     この HPL のパッチ済みバージョンは、オプションで実行中のパフォーマンスの情報を印刷します。また、指定された場合、実行を終了します。

   - -DENDEARLY -DASYOUGO2 (「検索時間を短縮するためのオプション」セクションを参照) を使用してコンパイルし、負のしきい値を使用して時間を短縮します (Top 500 にエントリーする場合、提出目的の最終的な実行で負のしきい値を使用しないでください !) HPL 1.0a 入力ファイル HPL.dat の 13 行でしきい値を設定することができます。

   - 問題の完了まで実行する場合は、-DASYOUGO (「検索時間を短縮するためのオプション」セクションを参照) を使用してください。

5. 迅速なパフォーマンス・フィードバックを使用し、最良のパフォーマンスが得られるまでステップ 3 と 4 を繰り返します。

検索時間を短縮するためのオプション

多くのノードで問題の完了まで実行すると、長い時間がかかります。MP LINPACK の検索空間も巨大です。実行する問題のサイズのみでなく、ブロックサイズの数、グリッドレイアウト、ステップの先読み、異なる因数分解方法の使用なども影響します。以前に得られた最良のパフォーマンスよりも、0.01% 遅くなったことを発見するためだけに大きな問題を最後まで実行しても、膨大な時間の浪費です。

検索時間が短くなる可能性のあるオプションは 3 つあります。

- -DASYOUGO
- -DENDEARLY
- -DASYOUGO2

  これらのオプションはパフォーマンスに影響を与えるため、慎重に使用してください。DGEMM の内部パフォーマンスを参照するには、-DASYOUGO2 および -DASYOUGO2_DISPLAY を使用してコンパイルします。パフォーマンスは約 0.2% 損なわれますが、多くの有用な DGEMM 情報が提供されます。

以前の HPL に戻すには、これらのオプションを定義しないで最初から再コンパイルします (”make arch=<arch> clean all” を実行してみてください)。

-DASYOUGO: 実行が進行するとともに、パフォーマンス・データを提供します。LU 分解が発生するため、パフォーマンスは常に開始時は高く、徐々に低くなります。ASYOUGO パフォーマンス評価は通常 (LU 分解により遅くなるため) 高めに評価されますが、問題が進行するとともにより正確になります。ステップの先読みが多いほど、最初の数は正確でなくなります。ASYOUGO は MP LINPACK が実行する LU 分解を含めて評価しようとするため、実際に達成された DGEMM パフォーマンスを測定する ASYOUGO2 と比較して高めに評価されます。ASYOUGO の出力は ASYOUGO2 が提供する情報のサブセットであることに注意してください。このため、出力の詳細は、-DASYOUGO2 オプションの説明を参照してください。

-DENDEARLY: いくつかのステップの後に問題を終了します。このため、モニターしないで 10 から 20 程度の HPL をセットアップして実行し、最も速かった HPL のみを完了させることができます。-DENDEARLY は -DASYOUGO を仮定します。手間ではありませんが、両方定義する必要はありません。問題が早く終了するため、ENDEARLY をテストする場合は、HPL.dat のしきい値を負の数に設定することを推奨します。問題が早く終了する場合に残差を確認する意味はありません。-DENDEARLY を使用する場合、-DASYOUGO2 を使用してコンパイルしたほうが良い場合もあります。

-DENDEARLY の仕様を知っておく必要があります。

- -DENDEARLY は、ブロックサイズで DGEMM の反復を数回行った後、問題を停止します (ブロックサイズが大きいほど、得られる情報も多くなります)。5 または 6 のアップデートのみ印刷します (-DASYOUGO は問題を完了する前に 46 程度の出力を印刷します)。
- -DASYOUGO と -DENDEARLY のパフォーマンスは常に 1 つの速度で開始され、ゆっくり増加した後、終了に向かって速度が落ちます (LU 分解が行われるため)。-DENDEARLY は通常、速度が落ちる前に終了します。
- -DENDEARLY は、HPL エラーとともに問題を早く終了します。問題は完了していないため、見つからない残差 (間違っている) を無視する必要があります。しかし、初期のパフォーマンスは確認できるため、パフォーマンスが良い場合は -DENDEARLY を指定しないで問題を最後まで実行します。エラーチェックを回避するには、HPL.dat に含まれている HPL のしきい値パラメーターを負の数にしてください。
- -DENDEARLY は早く終了するため、HPL は問題が完了したと解釈し、問題が完了したものとして Gflop 評価を計算します。この誤った高い評価は無視してください。
- より大きな問題では、精度はより高くなります。-DENDEARLY が返す最後のアップデートは、問題の完了まで実行したときのアップデートに近くなります。-DENDEARLY は、小さな問題では近似が不十分です。この理由により、ENDEARLY は ASYOUGO2 と組み合わせて使用することを推奨します。ASYOUGO2 は実際の DGEMM パフォーマンスを報告するため、開始した問題への近似がより近くなります。

インテル ® コンパイラーを使用した場合、最もよく知られているインテル ® Itanium® 2 プロセッサー用のコンパイルオプションは、次のようになります。

```
-O2 -ipo -ipo_obj -ftz -IPF_fltacc -IPF_fma -unroll -w -tpp2
```

-DASYOUGO2: 詳細な単一ノードの DGEMM パフォーマンス情報を提供します。すべての DGEMM 呼び出しをキャプチャーして (Fortran BLAS を使用している場合)、データを記録します。このため、ルーチンには侵入型のオーバーヘッドが存在します。非侵入型の -DASYOUGO とは異なり、-DASYOUGO2 は、パフォーマンスをモニターするため DGEMM の呼び出しごとに中断します。たとえパフォーマンスへの影響が 0.1% 未満であることがわかっていても、大きな問題ではこのオーバーヘッドに注意する必要があります。

次に、ASYOUGO2 出力のサンプルを示します (最初の 3 つの非侵入数は ASYOUGO および ENDEARLY の説明を参照してください)。

```
Col=001280 Fract=0.050 Mflops=42454.99 (DT= 9.5 DF= 34.1
DMF=38322.78)
```

問題サイズは *N*=16000 で、ブロックサイズは 128 でした。10 ブロック、つまり 1280 列を処理した後、出力は画面に送られました。ここで、完了した列の小数は 1280/16000=0.08 です。行列分解により、約 20 の出力がさまざまな場所に印刷されます (`fractions 0.005,0.010,0.015,0.02,0.025,0.03,0.035, 0.04,0.045,0.05,0.055,0.06,0.065,0.07,0.075,0.080,0.085,0.09,0.095,.10,...,.195,.295,.395,...,.895`)。しかし、ここでは比較のために問題サイズが非常に小さくブロック数が非常に大きいため、`0.045` の値を印刷するとすぐに、列の小数である 0.08 が見つかりました。非常に大きな問題では、小数の数はより正確になります。上記の 46 を超える数は印刷されません。このため、アップデートの数はより小さな問題では 46 よりも少なく、より大きな問題では正確に 46 になります。

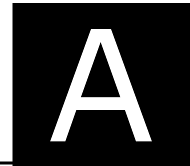
`Mflops` は、LU 分解が完了した 1280 列に基づく評価です。しかし、ステップの先読みが行われると、出力が行われるときに作業が実際に完了していない場合があります。しかし、これは同一の実行を比較するためには良い評価です。

括弧で囲まれている 3 つの数は、侵入型 `ASYOUGO2` のアドインです。`DT` は、プロセッサー 0 が `DGEMM` で費やした合計時間 (単位は秒) です。`DF` は、1 つのプロセッサーによって `DGEMM` で実行された処理の数 (単位は 10 億) です。したがって、プロセッサー 0 の `DGEMM` でのパフォーマンス (Gflops) は常に `DF/DT` になります。LU flops の数の代わりに `DGEMM` flops の数を基本として使用し、`DMF` を調べることで、実行のパフォーマンスの下限がわかります (`Mflops` はグローバル LU 時間を使用しますが、HPL のノード (0,0) のみは任意の出力を返すため、`DGEMM` flops は問題がノード間で平等に分散されているという仮定の下で計算されます)。

上記のパフォーマンス監視ツールを使用して異なる `HPL.dat` 入力を比較する場合、LU を使用したときのパフォーマンス低下のパターンは入力に敏感であることに注意してください。例えば、非常に小さな問題を実行した場合、初期値から終了値までのパフォーマンス低下は非常に急速です。より大きな問題では、パフォーマンス低下は少なくなるため、最初のいくつかのパフォーマンス値を使用して問題サイズの違い (例えば、7000000 と 701000) を評価しても安全です。パフォーマンス低下に影響を与える別の要因は、グリッドの次元 (P および Q) です。大きな問題では、P と Q が値でほぼ等しい場合、最初の数ステップからのパフォーマンス低下が少なくなる傾向があります。ブロードキャスト型のような大量のパラメーターを利用するように変更することで、最終的なパフォーマンスに非常に近いパフォーマンスを最初の数ステップで決定することができます。

これらのツールを使用すると、さまざまな量のデータをテストすることができます。

インテル ® マス・カーネル・ライブラリー言語インターフェイスのサポート



次の表は、各関数領域用にインテル ® マス・カーネル・ライブラリー (インテル ® MKL) が提供する言語インターフェイスを示しています。ただし、インテル ® MKL ルーチンは混在言語プログラミングを使用してほかの言語から呼び出すこともできます。例えば、Fortran ルーチンを C/C++ から呼び出す方法は、第 7 章の「混在言語プログラミングとインテル ® MKL」セクションを参照してください。

表 A-1 インテル ® MKL 言語インターフェイスのサポート

関数領域	Fortran 77 インターフェイス	Fortran 90/95 インターフェイス	C/C++ インターフェイス
Basic Linear Algebra Subprograms (BLAS)	+	+	CBLAS 利用
スパース BLAS レベル 1	+	+	CBLAS 利用
スパース BLAS レベル 2 およびレベル 3	+	+	+
連立線形方程式を解くための LAPACK ルーチン	+	+	
最小二乗問題、固有値ならびに特異値問題、およびシルベスター式を解くための LAPACK ルーチン	+	+	
補助 LAPACK ルーチン	+		
ScaLAPACK ルーチン	+		
PARDISO	+		+
その他の直接法および反復法スパース・ソルバー・ルーチン	+	+	+
ベクトル数学ライブラリー (VML) 関数		+	+
ベクトル・スタティスティカル・ライブラリー (VSL) 関数		+	+
フーリエ変換関数 (FFT)		+	+
クラスター FFT 関数		+	+
区間ソルバールーチン	+		
三角変換ルーチン		+	+
高速ポアソン、ラプラス、およびヘルムホルツ・ソルバー (ポアソン・ライブラリー) ルーチン		+	+
最適化 (Trust-Region) ソルバールーチン	+	+	+

サードパーティー・インターフェイスのサポート

本付録では、インテル® マス・カーネル・ライブラリー (インテル® MKL) がサポートする特定のインターフェイスについて簡単に説明します。

GMP* 関数

インテル MKL に実装されている GMP 数学関数には、任意精度の整数演算が含まれています。これらの関数のインターフェイスは、GMP (GNU Multiple Precision) 演算ライブラリーと互換性があります。

GMP ライブラリーを現在使用している場合、`mkl_gmp.h` をインクルードするようにプログラムの `INCLUDE` ステートメントを修正する必要があります。

FFTW インターフェイスのサポート

インテル® MKL は、インテル® MKL フーリエ変換関数を呼び出すために使用される、2 つのラッパー・コレクション (FFTW インターフェイスの上部構造) を提供します。これらのコレクションはそれぞれ、FFTW バージョン 2.x と 3.x に対応していて、インテル® MKL バージョン 7.0 以降で利用できます。

これらのラッパーの目的は、現在 FFTW を使用するプログラムの開発者が、プログラムのソースコードを変更することなく、インテル® MKL フーリエ変換を使用してパフォーマンスを向上できるようにすることです。FFTW 2.x ラッパーの使用についての詳細は 『*FFTW to Intel® MKL Wrappers Technical User Notes for FFTW 2.x*』(`fftw2xmkl_notes.htm`)、FFTW 3.x ラッパーの使用についての詳細は 『*FFTW to Intel® MKL Wrappers Technical User Notes for FFTW 3.x*』(`fftw3xmkl_notes.htm`) をそれぞれ参照してください。

索引

B

BLAS
- C から呼び出し 7-5
- Fortran-95 インターフェイス 7-2

C

CBLAS 7-5
CBLAS、コードの例 7-8
C、LAPACK、BLAS、CBLAS の呼び出し 7-4

E

Eclipse CDT、構成 4-2

F

FFT インターフェイス
- MKL_LONG 型 3-6
- 最適化基数 6-13
- スレッド化のヒント 6-10

FFT 関数、データのアライメント 6-11
FFTW インターフェイスのサポート B-1
Fortran-95、LAPACK と BLAS のインターフェイス 7-2

G

GMP 演算ライブラリー B-1
GMP 数学関数 B-1

H

HT テクノロジー、→ハイパースレッディング・テクノロジー

I

ILP64 プログラミング、サポート 3-4

J

Java の例 7-8

L

LAPACK
- C から呼び出し 7-4
- Fortran-95 インターフェイス 7-2
- 圧縮ルーチンのパフォーマンス 6-10

LINPACK ベンチマーク 10-1

M

MP LINPACK ベンチマーク 10-3

O

OpenMP
- 互換ランタイム・コンパイラー・ライブラリー 5-6
- レガシー・ランタイム・コンパイラー・ライブラリー 5-6

R

RTL 7-3
RTL レイヤー 3-3

S

ScaLAPACK、リンク 9-1

あ

アフィニティー・マスク 6-12

い

インストール、確認 2-1
インターフェイス・レイヤー 3-3

か

開発環境の構成 4-1
  Eclipse CDT 4-2
  ライブラリー名の再定義 4-5
カスタム共有オブジェクト 5-10
  関数のリストの指定 5-11
  構築 5-10
  メイクファイル・パラメーターの指定 5-10
環境変数、設定 4-1

く

クラスター FFT、リンク 9-1
クラスター・ソフトウェア 9-1
  リンク構文 9-1
  例のリンク 9-3

け

計算レイヤー 3-4
言語インターフェイスのサポート A-1
  Fortran-95 インターフェイス 7-2
  言語固有インターフェイス 7-1

こ

構成ファイル 4-4
構成、開発環境 4-1
構文
  リンク、一般 5-3
  リンク、クラスター・ソフトウェア 9-1
コーディング
  混在言語の呼び出し 7-5
  データのアライメント 8-1
  パフォーマンスを向上する手法 6-10
互換 OpenMP ランタイム・コンパイラー・ライブラリー 5-6
混在言語プログラミング 7-4
コンパイラー依存の関数 7-3
コンパイラー・サポート RTL レイヤー 3-4

さ

サポート , テクニカル 1-1
サポートするコンパイラー 2-2

し

使用法 1-1

す

数値計算の安定性 8-1
スタティック・リンク 5-1
スレッド化
  インテル® MKL コントロール 6-6
  環境変数と関数 6-6
  競合の回避 6-3
  →スレッド数
スレッド数
  設定する手法 6-2
スレッド化レイヤー 3-4
スレッド数
  OpenMP 環境変数を使用した設定 6-3
  クラスター用の設定 9-2
  ランタイムの変更 6-4

た

ダイナミック・リンク 5-1
対象読者 1-2
ダミー・ライブラリー 3-18

て

ディレクトリー構造
  高レベル 3-1
  詳細 3-10
  ドキュメント 3-19
テクニカルサポート 1-1

と

ドキュメント 3-19

は

ハイパースレッディング・テクノロジー、構成のヒント 6-11
パフォーマンス 6-1
- LAPACK 圧縮ルーチンの～ 6-10
- 向上するためのハードウェアのヒント 6-11
- 向上するためのヒント 6-10
- 非正規化数 6-12
- マルチコア 6-12

ひ

非正規化数、パフォーマンス 6-12
表記の規則 1-3

へ

並列処理 6-1
並列パフォーマンス 6-3
ベンチマーク 10-1

ま

マルチコア・パフォーマンス 6-12

め

メモリー関数名の変更 6-14
メモリー関数、再定義 6-14
メモリー管理 6-13

も

モジュール、Fortran-95 7-3

よ

呼び出し
- C から Fortran 形式のルーチン 7-4
- C から複素 BLAS レベル 1 関数 7-6
- C で BLAS 関数 7-5
- C++ から複素 BLAS レベル 1 関数 7-7

ら

ライブラリー
- ランタイム、互換 OpenMP 5-6
- ランタイム、レガシー OpenMP 5-6

ライブラリー構造 3-1
ライブラリーの逐次バージョン 3-4
ライブラリー名の再定義、構成ファイル 4-5
ランタイム・ライブラリー 7-3
- 互換 OpenMP 5-6
- レガシー OpenMP 5-6

り

リンク 5-1
- ScaLAPACK 9-1
- 基本レイヤーモデル 5-4
- クラスター FFT 9-1
- 推奨 5-2
- スタティック 5-1
- ダイナミック 5-1
- デフォルトモデル 5-4

リンクコマンド
- 構文 5-3
- 例 5-7

リンクモデル、比較 5-2
リンク・ライブラリー 5-5

れ

レイヤー
- RTL 3-3
- インターフェイス 3-3
- 計算 3-4
- コンパイラー・サポート RTL 3-4
- スレッド化 3-4

レイヤーモデル 3-2
例、ScaLAPACK、クラスター FFT、リンク 9-3
レガシー OpenMP ランタイム・コンパイラー・ライブラリー 5-6