Intel® Math Kernel Library for Linux* OS

User's Guide

MKL 10.3 - Linux OS*

Document Number: 315930-012US

Legal Information

Legal Information

Introducing the Intel(R) Math Kernel Library

The Intel(R) Math Kernel Library (Intel(R) MKL) enables improving performance of scientific, engineering, and financial software that solves large computational problems. Intel MKL provides a set of linear algebra routines, fast Fourier transforms, as well as vectorized math and random number generation functions, all optimized for the latest Intel(R) processors, including processors with multiple cores (see the *Intel(R) MKL Release Notes* for the full list of supported processors). Intel MKL also performs well on non-Intel processors.

Intel MKL provides the following major functionality:

- Linear algebra, implemented in LAPACK (solvers and eigensolvers) plus level 1, 2, and 3 BLAS, offering the vector, vector-matrix, and matrix-matrix operations needed for complex mathematical software. If you prefer the FORTRAN 90/95 programming language, you can call LAPACK driver and computational subroutines through specially designed interfaces with reduced numbers of arguments. C interface to LAPACK routines is also available.
- ScaLAPACK (SCAlable LAPACK) with its support functionality including the Basic Linear Algebra Communications Subprograms (BLACS) and the Parallel Basic Linear Algebra Subprograms (PBLAS). Available with Intel MKL for Linux* and Windows* operating systems.
- Direct sparse solver, an iterative sparse solver, and a supporting set of sparse BLAS (level 1, 2, and 3) for solving sparse systems of equations.
- Multidimensional discrete Fourier transforms (1D, 2D, 3D) with a mixed radix support (not limited to sizes of powers of 2). Distributed versions of these functions are provided for use on clusters on the Linux* and Windows* operating systems.
- A set of vectorized transcendental functions called the Vector Math Library (VML). For most of the supported processors, the Intel MKL VML functions offer greater performance than the libm (scalar) functions, while keeping the same high accuracy.
- The Vector Statistical Library (VSL), which offers high performance vectorized random number generators for several probability distributions, convolution and correlation routines, and summary statistics functions.

Intel MKL is thread-safe and extensively threaded using the OpenMP* technology.

For details see the *Intel(R) MKL Reference Manual*.

Optimization Notice
The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors. While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3

Optimization Notice
(Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Intel recommends that you evaluate other library products to determine which best meets your requirements.

Getting Help and Support

Intel provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more. Visit the Intel(R) MKL support website at http://www.intel.com/software/products/support/.

The Intel(R) Math Kernel Library documentation integrates into the Eclipse* integrated development environment (IDE). See Getting Assistance for Programming in the Eclipse* IDE.

Notational Conventions

The following term is used in reference to the operating system.

Linux* OS	This term refers to information that is valid on all supported Linux* operating systems.

The following notations are used to refer to Intel(R) Math Kernel Library (Intel(R) MKL) directories.

<*`mkl directory`*>	The main directory where Intel MKL is installed. Replace this placeholder with the specific pathname in the configuring, linking, and building instructions. The default value of this placeholder is `/opt/intel/composerxe-2011.y.xxx/mkl`, where `y` is the release-update number and `xxx` is the package number.
<*`Composer XE directory`*>	The installation directory for the Intel(R) C++ Composer XE 2011 or Intel(R) Fortran Composer XE 2011. The default value of this placeholder is `/opt/intel/composerxe-2011.y.xxx`.

The following font conventions are used in this document.

Italic	Italic is used for emphasis and also indicates document names in body text, for example: see *Intel MKL Reference Manual*.
`Monospace lowercase`	Indicates filenames, directory names and pathnames, for example: `libmkl_core.a` `/opt/intel/composerxe-2011.0.004`

`Monospace lowercase mixed with uppercase`	Indicates:
	• Commands and command-line options, for example,
	`icc myprog.c -L$MKLPATH -I$MKLINCLUDE -lmkl -liomp5 -lpthread`
	• C/C++ code fragments, for example,
	`a = new double [SIZE*SIZE];`

`UPPERCASE MONOSPACE`	Indicates system variables, for example, `$MKLPATH`.
`Monospace italic`	Indicates a parameter in discussions: • Routine parameters, for example, *`lda`* • Makefile parameters, for example, *`functions_list`* When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value, for example, <*`mkl directory`*>. Substitute one of these items for the placeholder.
`[ items ]`	Square brackets indicate that the items enclosed in brackets are optional.

`{ item | item }`	Braces indicate that only one of the items listed between braces should be selected. A vertical bar (|) separates the items.

Contents

Chapter 10: LINPACK and MP LINPACK Benchmarks

Appendix A: Intel(R) Math Kernel Library Language Interfaces Support

Appendix B: Support for Third-Party Interfaces

Appendix C: Directory Structure in Detail

Index

Overview

1

Document Overview

The Intel(R) Math Kernel Library (Intel(R) MKL) User's Guide provides *usage information* for the library. The usage information covers the organization, configuration, performance, and accuracy of Intel MKL, specifics of routine calls in mixed-language programming, linking, and more.

This guide describes OS-specific usage of Intel MKL, along with OS-independent features. It contains usage information for all Intel MKL function domains.

This User's Guide provides the following information:

- Describes post-installation steps to help you start using the library
- Shows you how to configure the library with your development environment
- Acquaints you with the library structure
- Explains how to link your application with the library and provides simple usage scenarios
- Describes how to code, compile, and run your application with Intel MKL

This guide is intended for Linux OS programmers with beginner to advanced experience in software development.

See Also

- Intel MKL Function Domains

What's New

This User's Guide documents the Intel(R) Math Kernel Library (Intel(R) MKL) 10.3 Update 1.

Starting with release 10.3,

- Intel MKL is delivered as part of the Intel(R) C++ Composer XE and Intel(R) Fortran Composer XE
- The library discontinued support of the IA-64 architecture

Respective changes in the Intel MKL structure have been documented in the user's guide.

The following new features of the library have been documented:

- Single Dynamic Library interface and use of this interface by the custom shared object builder
- New Fortran 95 interface modules
- Parameters of the environment setting scripts
- Use of C interface to LAPACK

The document has been considerably restructured.

Related Information

To reference how to use the library in your application, use this guide in conjunction with the following documents:

- The *Intel(R) Math Kernel Library Reference Manual*, which provides *reference* information on routine functionalities, parameter descriptions, interfaces, calling syntaxes, and return values.
- The *Intel(R) Math Kernel Library for Linux* OS Release Notes*.

Getting Started

2

Optimization Notice

The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

Checking Your Installation

After installing the Intel(R) Math Kernel Library (Intel(R) MKL), verify that the library is properly installed and configured:

1. Intel MKL installs in `<Composer XE directory>`.

   Check that the subdirectory of `<Composer XE directory>` referred to as `<mkl directory>` was created. By default, `/opt/intel/composerxe-2011.y.xxx/mkl`, where `y` is the release-update number and `xxx` is the package number.
2. If you want to keep multiple versions of Intel MKL installed on your system, update your build scripts to point to the correct Intel MKL version.
3. Check that the following files appear in the `<mkl directory>/bin` directory and its subdirectories:

   `mklvars.sh`

   `mklvars.csh`

   `ia32/mklvars_ia32.sh`

   `ia32/mklvars_ia32.csh`

   `intel64/mklvars_intel64.sh`

   `intel64/mklvars_intel64.csh`

   Use these files to assign Intel MKL-specific values to several environment variables.

4. To understand how the Intel MKL directories are structured, see Intel(R) Math Kernel Library Structure.

See Also

- Setting Environment Variables

Setting Environment Variables

See Also

- Setting the Number of Threads Using an OpenMP* Environment Variable

Scripts to Set Environment Variables

When the installation of the Intel(R) Math Kernel Library (Intel(R) MKL) for Linux* OS is complete, set the `INCLUDE`, `MKLROOT`, `LD_LIBRARY_PATH`, `MANPATH`, `LIBRARY_PATH`, `CPATH`, `FPATH`, and `NLSPATH` environment variables in the command shell using one of the script files in the `bin` subdirectory of the Intel MKL installation directory. Choose the script corresponding to your system architecture and command shell as explained in the following table:

Architecture	Shell	Script File
IA-32	C	`ia32/mklvars_ia32.csh`
IA-32	Bash and Bourne (sh)	`ia32/mklvars_ia32.sh`
Intel(R) 64	C	`intel64/mklvars_intel64.csh`
Intel(R) 64	Bash and Bourne (sh)	`intel64/mklvars_intel64.sh`
IA-32 and Intel(R) 64	C	`mklvars.csh`
IA-32 and Intel(R) 64	Bash and Bourne (sh)	`mklvars.sh`

Running the Scripts

The scripts accept parameters to specify the following:

- Architecture.
- Addition of a path to Fortran 95 modules precompiled with the Intel(R) Fortran compiler to the `FPATH` environment variable. Supply this parameter only if you are using the Intel(R) Fortran compiler.
- Interface of the Fortran 95 modules. This parameter is needed only if you requested addition of a path to the modules.

Usage and values of these parameters depend on the script name (regardless of the extension). The following table lists values of the script parameters.

Script	Architecture (required, when applicable)	Addition of a Path to Fortran 95 Modules (optional)	Interface (optional)
mklvars_ia32	n/a[†]	mod	n/a
mklvars_intel64	n/a	mod	lp64, default ilp64
mklvars	ia32 intel64	mod	lp64, default ilp64

[†]Not applicable.

For example:

- The command `mklvars_ia32.sh` sets environment variables for the IA-32 architecture and adds no path to the Fortran 95 modules.
- The command `mklvars_intel64.sh mod ilp64` sets environment variables for the Intel(R) 64 architecture and adds the path to the Fortran 95 modules for the ILP64 interface to the `FPATH` environment variable.
- The command `mklvars.sh intel64 mod` sets environment variables for the Intel(R) 64 architecture and adds the path to the Fortran 95 modules for the LP64 interface to the `FPATH` environment variable.

NOTE. Supply the parameter specifying the architecture first, if it is needed. Values of the other two parameters can be listed in any order.

See Also

- High-level Directory Structure
- Intel(R) MKL Interface Libraries and Modules
- Fortran 95 Interfaces to LAPACK and BLA
- Setting the Number of Threads Using an OpenMP* Environment Variable

Automating the Process of Setting Environment Variables

To automate setting of the `INCLUDE, MKLROOT, LD_LIBRARY_PATH, MANPATH, LIBRARY_PATH, CPATH, FPATH`, and `NLSPATH` environment variables, add `mklvars*.*sh` to your shell profile so that each time you login, the script automatically executes and sets the paths to the appropriate Intel(R) Math Kernel Library (Intel(R) MKL) directories. To do this, with a local user account, edit the following files by adding the appropriate script to the path manipulation section right before exporting variables:

Shell	Files	Commands
bash	`~/.bash_profile`, `~/.bash_login` or `~/.profile`	`# setting up MKL environment for bash` `. <absolute_path_to_installed_MKL>/bin [/<arch>]/mklvars[<arch>].sh [<arch>] [mod] [lp64|ilp64]`

Shell	Files	Commands
sh	`~/.profile`	`# setting up MKL environment for sh` `. <absolute_path_to_installed_MKL>/bin [/<arch>]/mklvars[<arch>].sh [<arch>] [mod] [lp64|ilp64]`
csh	`~/.login`	`# setting up MKL environment for sh` `. <absolute_path_to_installed_MKL>/bin [/<arch>]/mklvars[<arch>].csh [<arch>] [mod] [lp64|ilp64]`

In the above commands, replace *<arch>* with `ia32` or `intel64`.

If you have super user permissions, add the same commands to a general-system file in `/etc/profile` (for bash and sh) or in `/etc/csh.login` (for csh).

CAUTION. Before uninstalling Intel MKL, remove the above commands from all profile files where the script execution was added. Otherwise you may experience problems logging in.

See Also

- Scripts to Set Environment Variables

Compiler Support

Intel(R) Math Kernel Library (Intel(R) MKL) supports compilers identified in the *Release Notes*. However, the library has been successfully used with other compilers as well.

Intel MKL provides a set of include files to simplify program development by specifying enumerated values and prototypes for the respective functions. Calling Intel MKL functions from your application without an appropriate include file may lead to incorrect behavior of the functions.

See Also

- Intel(R) MKL Include Files

Using Code Examples

The Intel(R) Math Kernel Library (Intel(R) MKL) package includes code examples, located in the `examples` subdirectory of the installation directory. Use the examples to determine:

- Whether Intel MKL is working on your system
- How you should call the library
- How to link the library

The examples are grouped in subdirectories mainly by Intel MKL function domains and programming languages. For example, the `examples/spblas` subdirectory contains a makefile to build the Sparse BLAS examples, and the `examples/vmlc` subdirectory contains the makefile to build the C VML examples. Source code for the examples is in the next-level `sources` subdirectory.

See Also

- High-level Directory Structure

Using the Web-based Linking Advisor

Use the Intel(R) Math Kernel Library (Intel(R) MKL) Linking Advisor to determine the libraries and options to specify on your link or compilation line.

The tool is available at http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor.

The Linking Advisor requests information about your system and on how you intend to use Intel MKL (link dynamically or statically, use threaded or sequential mode, etc.). The tool automatically generates the appropriate link line for your application.

See Also

- Linking Your Application with the Intel(R) Math Kernel Library
- Examples for Linking with ScaLAPACK and Cluster FFT

What You Need to Know Before You Begin Using the Intel(R) Math Kernel Library

Target platform	Identify the architecture of your target machine: • IA-32 or compatible • Intel(R) 64 or compatible **Reason:** Because Intel MKL libraries are located in directories corresponding to your particular architecture (see Architecture Support), you should provide proper paths on your link lines (see Linking Examples). To configure your development environment for the use with Intel MKL, set your environment variables using the script corresponding to your architecture (see Setting Environment Variables for details).
Mathematical problem	Identify all Intel MKL function domains that you require: • BLAS • Sparse BLAS • LAPACK • PBLAS • ScaLAPACK • Sparse Solver routines • Vector Mathematical Library functions (VML) • Vector Statistical Library functions • Fourier Transform functions (FFT) • Cluster FFT

	• Trigonometric Transform routines • Poisson, Laplace, and Helmholtz Solver routines • Optimization (Trust-Region) Solver routines • GMP* arithmetic functions **Reason:** The function domain you intend to use narrows the search in the *Reference Manual* for specific routines you need. Additionally, if you are using the Intel MKL cluster software, your link line is function-domain specific (see Working with the Cluster Software). Coding tips may also depend on the function domain (see Tips and Techniques to Improve Performance).
Programming language	Intel MKL provides support for both Fortran and C/C++ programming. Identify the language interfaces that your function domains support (see Intel(R) Math Kernel Library Language Interfaces Support). **Reason:** Intel MKL provides language-specific include files for each function domain to simplify program development (see Language Interfaces Support, by Function Domain). For a list of language-specific interface libraries and modules and an example how to generate them, see also Using Language-Specific Interfaces with Intel(R) MKL.
Range of integer data	If your system is based on the Intel 64 architecture, identify whether your application performs calculations with large data arrays (of more than 2^{31}-1 elements). **Reason:** To operate on large data arrays, you need to select the ILP64 interface, where integers are 64-bit; otherwise, use the default, LP64, interface, where integers are 32-bit (see Using the ILP64 Interface vs. LP64 Interface).
Threading model	Identify whether and how your application is threaded: • Threaded with the Intel(R) compiler • Threaded with a third-party compiler • Not threaded **Reason:** The compiler you use to thread your application determines which threading library you should link with your application. For applications threaded with a third-party compiler you may need to use Intel MKL in the sequential mode (for more information, see Sequential Mode of the Library and Linking with Threading Libraries).
Number of threads	Determine the number of threads you want Intel MKL to use. **Reason:** Intel MKL is based on the OpenMP* threading. By default, the OpenMP* software sets the number of threads that Intel MKL uses. If you need a different number, you have to set it yourself using one of the available mechanisms. For more information, see Using Parallelism of the Intel(R) Math Kernel Library.
Linking model	Decide which linking model is appropriate for linking your application with Intel MKL libraries: • Static • Dynamic **Reason:** The link line syntax and libraries for static and dynamic linking are different. For the list of link libraries for static and dynamic models, linking examples, and other relevant topics, like how to save disk space by creating a custom dynamic library, see Linking Your Application with the Intel(R) Math Kernel Library.

MPI used	Decide what MPI you will use with the Intel MKL cluster software. You are strongly encouraged to use Intel(R) MPI 3.2 or later. **Reason:** To link your application with ScaLAPACK and/or Cluster FFT, the libraries corresponding to your particular MPI should be listed on the link line (see Working with the Cluster Software).

Structure of the Intel(R) Math Kernel Library

3

Optimization Notice
The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors. While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Intel recommends that you evaluate other library products to determine which best meets your requirements.

Architecture Support

Intel(R) Math Kernel Library (Intel(R) MKL) for Linux* OS provides two architecture-specific implementations. The following table lists the supported architectures and directories where each architecture-specific implementation is located.

Architecture	Location
IA-32 or compatible	`<mkl directory>/lib/ia32`
Intel(R) 64 or compatible	`<mkl directory>/lib/intel64`

See Also

- High-level Directory Structure
- Detailed Structure of the IA-32 Architecture Directory lib/ia32
- Detailed Structure of the Intel(R) 64 Architecture Directory lib/intel64

High-level Directory Structure

Directory	Contents
<mkl directory>	Installation directory of the Intel(R) Math Kernel Library (Intel(R) MKL). By default, `/opt/intel/composerxe-2011.y.xxx/mkl`, where y is the release-update number, and `xxx` is the package number.
Subdirectories of *<mkl directory>*	
`bin`	Scripts to set environmental variables in the user shell
`bin/ia32`	Shell scripts for the IA-32 architecture
`bin/intel64`	Shell scripts for the Intel(R) 64 architecture
`benchmarks/linpack`	Shared-memory (SMP) version of the LINPACK benchmark
`benchmarks/mp_linpack`	Message-passing interface (MPI) version of the LINPACK benchmark
`examples`	Examples directory. Each subdirectory has source and data files
`include`	INCLUDE files for the library routines, as well as for tests and examples
`include/ia32`	Fortran 95 .mod files for the IA-32 architecture and Intel(R) Fortran compiler
`include/intel64/lp64`	Fortran 95 .mod files for the Intel(R) 64 architecture, Intel Fortran compiler, and LP64 interface
`include/intel64/ilp64`	Fortran 95 .mod files for the Intel(R) 64 architecture, Intel Fortran compiler, and ILP64 interface
`interfaces/blas95`	Fortran 95 interfaces to BLAS and a makefile to build the library
`interfaces/fftw2x_cdft`	MPI FFTW 2.x interfaces to the Intel MKL Cluster FFTs
`interfaces/fftw3x_cdft`	MPI FFTW 3.x interfaces to the Intel MKL Cluster FFTs
`interfaces/fftw2xc`	FFTW 2.x interfaces to the Intel MKL FFTs (C interface)
`interfaces/fftw2xf`	FFTW 2.x interfaces to the Intel MKL FFTs (Fortran interface)
`interfaces/fftw3xc`	FFTW 3.x interfaces to the Intel MKL FFTs (C interface)
`interfaces/fftw3xf`	FFTW 3.x interfaces to the Intel MKL FFTs (Fortran interface)
`interfaces/lapack95`	Fortran 95 interfaces to LAPACK and a makefile to build the library
`lib/ia32`	Static libraries and shared objects for the IA-32 architecture
`lib/intel64`	Static libraries and shared objects for the Intel(R) 64 architecture

Directory	Contents
`tests`	Source and data files for tests
`tools/builder`	Tools for creating custom dynamically linkable libraries
`tools/plugins/ com.intel.mkl.help`	Eclipse* IDE plug-in with Intel MKL Reference Manual in WebHelp format. See `mkl_documentation.htm` for more information
Subdirectories of *`<Composer XE directory>`*. By default, `/opt/intel/composerxe-2011.y.xxx`.	
`Documentation/en_US/mkl`	Intel MKL documentation.
`man/en_US/man3`	Man pages for Intel MKL functions. No directory for man pages is created in locales other than en_US even if a directory for the localized documentation is created in the respective locales. For more information, see Contents of the Documentation Directories.

Layered Model Concept

Intel(R) Math Kernel Library (Intel(R) MKL) is structured to support multiple compilers and interfaces, different OpenMP* implementations, both serial and multiple threads, and a wide range of processors. Conceptually Intel MKL can be divided into distinct parts to support different interfaces, threading models, and core computations:

1. Interface Layer
2. Threading Layer
3. Computational Layer

You can combine Intel MKL libraries to meet your needs by linking with one library in each part layer-by-layer. Once the interface library is selected, the threading library you select picks up the chosen interface, and the computational library uses interfaces and OpenMP implementation (or non-threaded mode) chosen in the first two layers.

To support threading with different compilers, one more layer is needed, which contains libraries not included in Intel MKL:

- Compiler-support run-time libraries (RTL).

The following table provides more details of each layer.

Layer	Description
Interface Layer	Matches compiled code of your application with the threading and/or computational parts of the library. This layer provides: • LP64 and ILP64 interfaces. • Compatibility with compilers that return function values differently. • A mapping between single-precision names and double-precision names for applications using Cray*-style naming (SP2DP interface).

Layer	Description

	SP2DP interface supports Cray-style naming in applications targeted for the Intel 64 architecture and using the ILP64 interface. SP2DP interface provides a mapping between single-precision names (for both real and complex types) in the application and double-precision names in Intel MKL BLAS and LAPACK. Function names are mapped as shown in the following example for BLAS functions `?GEMM`:
	
	`SGEMM -> DGEMM`
	`DGEMM -> DGEMM`
	`CGEMM -> ZGEMM`
	`ZGEMM -> ZGEMM`
	Mind that no changes are made to double-precision names.

Threading Layer	This layer: • Provides a way to link threaded Intel MKL with different threading compilers. • Enables you to link with a threaded or sequential mode of the library. This layer is compiled for different environments (threaded or sequential) and compilers (from Intel, GNU*, and so on).
Computational Layer	Heart of Intel MKL. This layer has only one library for each combination of architecture and supported OS. The Computational layer accommodates multiple architectures through identification of architecture features and chooses the appropriate binary code at run time.
Compiler Support Run-time Libraries (RTL)	To support threading with Intel compilers, Intel MKL uses the compiler support RTL of the Intel(R) C++ Composer XE or Intel(R) Fortran Composer XE. To thread using third-party threading compilers, use libraries in the Threading layer or an appropriate compatibility library.

See Also

- Using the ILP64 Interface vs. LP64 Interface
- Linking Your Application with the Intel(R) Math Kernel Library
- Linking with Threading Libraries

Accessing the Intel(R) Math Kernel Library Documentation

Contents of the Documentation Directories

Most of Intel(R) Math Kernel Library (Intel(R) MKL) documentation is installed at `<`*`Composer XE directory`*`>/Documentation/<`*`locale`*`>/mkl`. For example, the documentation in English is installed at `<`*`Composer XE directory`*`>/Documentation/en_US/mkl`. However, some Intel MKL-related documents are installed one or two levels up. The following table lists MKL-related documentation.

File name	Comment
Files in `<`*`Composer XE directory`*`>/Documentation`	

File name	Comment
<*locale*>/clicense or <*locale*>/flicense	Common end user license for the Intel(R) C++ Composer XE 2011 or Intel(R) Fortran Composer XE 2011, respectively
mklsupport.txt	Information on package number for customer support reference
Contents of <*Composer XE directory*>/Documentation/<*locale*>/mkl	
redist.txt	List of redistributable files
mkl_documentation.htm	Overview and links for the Intel MKL documentation
mkl_manual/index.htm	Intel MKL Reference Manual in an uncompressed HTML format
Release_Notes.htm	Intel MKL Release Notes
mkl_userguide/index.htm	Intel MKL User's Guide in an uncompressed HTML format, this document

Viewing Man Pages

To access man pages for the Intel(R) Math Kernel Library (Intel(R) MKL), add the man pages directory to the MANPATH environment variable. If you performed the Setting Environment Variables step of the Getting Started process, this is done automatically.

To view the man page for an Intel MKL function, enter the following command in your command shell:

`man <function base name>`

In this release, <*function base name*> is the function name with omitted prefixes denoting data type, precision, or function domain.

Examples:

- For the BLAS function `ddot`, enter `man dot`
- For the ScaLAPACK function `pzgeql2`, enter `man pgeql2`
- For the FFT function `DftiCommitDescriptor`, enter `man CommitDescriptor`

NOTE. Function names in the `man` command are case-sensitive.

See Also

- High-level Directory Structure
- Setting Environment Variables

Linking Your Application with the Intel(R) Math Kernel Library

4

Optimization Notice
The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors. While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Intel recommends that you evaluate other library products to determine which best meets your requirements.

Linking Quick Start

To link with Intel(R) Math Kernel Library (Intel(R) MKL), choose one library from the Interface layer, one library from the Threading layer, one (and typically the only) library from the Computational layer, and, if necessary, add run-time libraries. The following table lists Intel MKL libraries to link with your application.

	Interface layer	**Threading layer**	**Computational layer**	**RTL**
IA-32 architecture, static linking	`libmkl_intel.a`	`libmkl_intel_thread.a`	`libmkl_core.a`	`libiomp5.so`
IA-32 architecture, dynamic linking	`libmkl_rt.so`			
Intel(R) 64 architecture, static linking	`libmkl_intel_lp64.a`	`libmkl_intel_thread.a`	`libmkl_core.a`	`libiomp5.so`
Intel(R) 64 architecture, dynamic linking	`libmkl_rt.so`			

For exceptions and alternatives to the libraries listed above, see Selecting Libraries to Link.

See Also

- Using the Web-based Linking Advisor
- Using the Single Dynamic Library Interface
- Listing Libraries on a Link Line
- Working with the Cluster Software

Listing Libraries on a Link Line

To link with Intel(R) Math Kernel Library, specify paths and libraries on the link line as shown below.

NOTE. The syntax below is for dynamic linking. For static linking, replace each library name preceded with "`-l`" with the path to the library file. For example, replace `-lmkl_core` with `$MKLPATH/libmkl_core.a`, where $MKLPATH is the appropriate user-defined environment variable.

```
<files to link>

-L<MKL path> -I<MKL include>
[-I<MKL include>/{ia32|intel64|{ilp64|lp64}}]

[-lmkl_blas{95|95_ilp64|95_lp64}]
[-lmkl_lapack{95|95_ilp64|95_lp64}]

[<cluster components>]

-lmkl_{intel|intel_ilp64|intel_lp64|intel_sp2dp|gf|gf_ilp64|gf_lp64}

-lmkl_{intel_thread|gnu_thread|pgi_thread|sequential}

-lmkl_core

-liomp5 [-lpthread] [-lm]
```

In case of static linking, enclose the cluster components, interface, threading, and computational libraries in grouping symbols (for example, `-Wl,--start-group $MKLPATH/libmkl_cdft_core.a $MKLPATH/libmkl_blacs_intelmpi_ilp64.a $MKLPATH/libmkl_intel_ilp64.a $MKLPATH/libmkl_intel_thread.a $MKLPATH/libmkl_core.a -Wl,--end-group`).

The order of listing libraries on the link line is essential, except for the libraries enclosed in the grouping symbols above.

See Also

- Using the Web-based Linking Advisor
- Selecting Libraries to Link
- Linking Examples
- Working with the Cluster Software

Selecting Libraries to Link

This section recommends which libraries to link against depending on your Intel MKL usage scenario and provides details of the linking.

Linking with Interface Libraries

Using the Single Dynamic Library Interface

Intel(R) Math Kernel Library (Intel(R) MKL) provides the Single Dynamic Library interface (SDL interface). It enables you to dynamically select the interface and threading layer for Intel MKL.

To use the SDL interface, put the only library on your link line: `libmkl_rt.so`. For example,

```
ic  application.c -lmkl_rt
```

Setting the Threading Layer

To set the threading layer at run time, use the `mkl_set_threading_layer` function or the `MKL_THREADING_LAYER` environment variable. The following table lists available threading layers along with values to be used to set each layer.

Threading Layer	Value of `MKL_THREADING_LAYER`	Value of the Parameter of `mkl_set_threading_layer`
Intel threading	`INTEL`	`MKL_THREADING_INTEL`
Sequential mode of Intel MKL	`SEQUENTIAL`	`MKL_THREADING_SEQUENTIAL`
GNU threading	`GNU`	`MKL_THREADING_GNU`
PGI threading	`PGI`	`MKL_THREADING_PGI`

If the `mkl_set_threading_layer` function is called, the environment variable `MKL_THREADING_LAYER` is ignored.

By default Intel threading is used.

See the *Intel MKL Reference Manual* for details of the `mkl_set_threading_layer` function.

Setting the Interface Layer

Available interfaces depend on the architecture of your system.

On systems based on the Intel(R) 64 architecture, LP64 and ILP64 interfaces are available. To set one of these interfaces at run time, use the `mkl_set_interface_layer` function or the `MKL_INTERFACE_LAYER` environment variable. The following table provides values to be used to set each interface.

Interface Layer	**Value of `MKL_INTERFACE_LAYER`**	**Value of the Parameter of `mkl_set_interface_layer`**
LP64	`LP64`	`MKL_INTERFACE_LP64`
ILP64	`ILP64`	`MKL_INTERFACE_ILP64`

If the `mkl_set_interface_layer` function is called, the environment variable `MKL_SET_INTERFACE_LAYER` is ignored.

By default the LP64 interface is used.

See the *Intel MKL Reference Manual* for details of the `mkl_set_interface_layer` function.

See Also

- Layered Model Concept
- Directory Structure in Detail

Using the ILP64 Interface vs. LP64 Interface

The Intel(R) Math Kernel Library (Intel(R) MKL) ILP64 libraries use the 64-bit integer type (necessary for indexing large arrays, with more than 2^{31}-1 elements), whereas the LP64 libraries index arrays with the 32-bit integer type.

The LP64 and ILP64 interfaces are implemented in the Interface layer. Link with the following interface libraries for the LP64 or ILP64 interface, respectively:

- `libmkl_intel_lp64.a` or `libmkl_intel_ilp64.a` for static linking
- `libmkl_intel_lp64.so` or `libmkl_intel_ilp64.so` for dynamic linking

The ILP64 interface provides for the following:

- Support large data arrays (with more than 2^{31}-1 elements)
- Enable compiling your Fortran code with the `-i8` compiler option

The LP64 interface provides compatibility with the previous Intel MKL versions because "LP64" is just a new name for the only interface that the Intel MKL versions lower than 9.1 provided. Choose the ILP64 interface if your application uses Intel MKL for calculations with large data arrays or the library may be used so in future.

Intel MKL provides the same include directory for the ILP64 and LP64 interfaces.

Compiling for LP64/ILP64

The table below shows how to compile for the ILP64 and LP64 interfaces:

Fortran	
Compiling for ILP64	`ifort -i8 -I<mkl directory>/include ...`
Compiling for LP64	`ifort -I<mkl directory>/include ...`

C or C++	
Compiling for ILP64	icc -DMKL_ILP64 -I<*mkl directory*>/include ...
Compiling for LP64	icc -I<*mkl directory*>/include ...

CAUTION. Linking of an application compiled with the -i8 or -DMKL_ILP64 option to the LP64 libraries may result in unpredictable consequences and erroneous output.

Coding for ILP64

You do not need to change existing code if you are not using the ILP64 interface.

To migrate to ILP64 or write new code for ILP64, use appropriate types for parameters of the Intel MKL functions and subroutines:

Integer Types	Fortran	C or C++
32-bit integers	INTEGER*4 or INTEGER(KIND=4)	int
Universal integers for ILP64/LP64: • 64-bit for ILP64 • 32-bit otherwise	INTEGER without specifying KIND	MKL_INT
Universal integers for ILP64/LP64: • 64-bit integers	INTEGER*8 or INTEGER(KIND=8)	MKL_INT64
FFT interface integers for ILP64/LP64	INTEGER without specifying KIND	MKL_LONG

Browsing the Intel MKL Include Files

The *Reference Manual* does not explain which integer parameters of a function become 64-bit and which remain 32-bit for ILP64. To get to know this, browse the include files, examples, and tests for the ILP64 interface details.

Some function domains that support only a Fortran interface provide header files for C/C++ in the include directory. Such *.h files enable using a Fortran binary interface from C/C++ code. These files can also be used to understand the ILP64 usage.

Limitations

All Intel MKL function domains support ILP64 programming with the following exceptions:

- FFTW interfaces to Intel MKL:

    - FFTW 2.x wrappers do not support ILP64.
    - FFTW 3.2 wrappers support ILP64 by a dedicated set of functions `plan_guru64`.

- GMP* arithmetic functions do not support ILP64.

See Also

- High-level Directory Structure
- Intel(R) MKL Include Files
- Language Interfaces Support, by Function Domain
- Layered Model Concept
- Directory Structure in Detail

Linking with Fortran 95 Interface Libraries

The `libmkl_blas95*.a` and `libmkl_lapack95*.a` libraries contain Fortran 95 interfaces for BLAS and LAPACK, respectively, which are compiler-dependent. In the Intel(R) Math Kernel Library (Intel(R) MKL) package, they are prebuilt for the Intel(R) Fortran compiler. If you are using a different compiler, build these libraries before using the interface.

See Also

- Fortran 95 Interfaces to LAPACK and BLAS
- Compiler-dependent Functions and Fortran 90 Modules

Linking with Threading Libraries

Sequential Mode of the Library

You can use Intel(R) Math Kernel Library (Intel(R) MKL) in a sequential (non-threaded) mode. In this mode, Intel MKL runs unthreaded code. However, it is thread-safe (except the LAPACK deprecated routine `?lacon`), which means that you can use it in a parallel region in your OpenMP* code. The sequential mode requires no compatibility OpenMP* run-time library and does not respond to the environment variable `OMP_NUM_THREADS` or its Intel MKL equivalents.

You should use the library in the sequential mode only if you have a particular reason not to use Intel MKL threading. The sequential mode may be helpful when using Intel MKL with programs threaded with some non-Intel compilers or in other situations where you need a non-threaded version of the library (for instance, in some MPI cases). To set the sequential mode, in the Threading layer, choose the `*sequential.*` library.

Add the POSIX threads library `(pthread)` to your link line for the sequential mode because the `*sequential.*` library depends on `pthread`.

See Also

- Directory Structure in Detail
- Using Parallelism of the Intel(R) Math Kernel Library
- Avoiding Conflicts in the Execution Environment
- Linking Examples

Selecting Libraries in the Threading Layer and RTL

Several compilers that Intel(R) Math Kernel Library (Intel(R) MKL) supports use the OpenMP* threading technology. Intel MKL supports implementations of the OpenMP* technology that these compilers provide. To make use of this support, you need to link with the appropriate library in the Threading Layer and Compiler Support Run-time Library (RTL).

Threading Layer

Each Intel MKL threading library contains the same code compiled by the respective compiler (Intel, gnu and PGI* compilers on Linux OS).

RTL

This layer includes `libiomp`, the compatibility OpenMP* run-time library of the Intel compiler. In addition to the Intel compiler, `libiomp` provides support for one additional threading compiler on Linux OS (GNU). That is, a program threaded with a GNU compiler can safely be linked with Intel MKL and `libiomp`.

The table below helps explain what threading library and RTL you should choose under different scenarios when using Intel MKL (static cases only):

Compiler	Application Threaded?	Threading Layer	RTL Recommended	Comment
Intel	Does not matter	`libmkl_intel_thread.a`	`libiomp5.so`	
PGI	Yes	`libmkl_pgi_thread.a` or `libmkl_sequential.a`	PGI* supplied	Use of `libmkl_sequential.a` removes threading from Intel MKL calls.
PGI	No	`libmkl_intel_thread.a`	`libiomp5.so`	
PGI	No	`libmkl_pgi_thread.a`	PGI* supplied	
PGI	No	`libmkl_sequential.a`	None	
gnu	Yes	`libmkl_gnu_thread.a`	`libiomp5.so` or GNU OpenMP run-time library	`libiomp5` offers superior scaling performance.
gnu	Yes	`libmkl_sequential.a`	None	
gnu	No	`libmkl_intel_thread.a`	`libiomp5.so`	
other	Yes	`libmkl_sequential.a`	None	
other	No	`libmkl_intel_thread.a`	`libiomp5.so`	

Linking with Computational Libraries

If you are not using the Intel(R) Math Kernel Library (Intel(R) MKL) cluster software, you need to link your application with only one computational library:

- `libmkl_core.a` in case of static linking
- `libmkl_core.so` in case of dynamic linking

ScaLAPACK and Cluster FFT require more computational libraries. The following table lists computational libraries that you must list on the link line for these function domains.

Function domain	IA-32 Architecture, Static Linking	IA-32 Architecture, Dynamic Linking	Intel(R) 64 Architecture, Static Linking	Intel(R) 64 Architecture, Dynamic Linking
ScaLAPACK[1]	`libmkl_scalapack_core.a` `libmkl_core.a`	`libmkl_scalapack_core.so` `libmkl_core.so`	See below	See below
ScaLAPACK, LP64 interface[1]	n/a[2]	n/a	`libmkl_scalapack_lp64.a` `libmkl_core.a`	`libmkl_scalapack_lp64.so` `libmkl_core.so`
ScaLAPACK, ILP64 interface[1]	n/a	n/a	`libmkl_scalapack_ilp64.a` `libmkl_core.a`	`libmkl_scalapack_ilp64.so` `libmkl_core.so`
Cluster Fourier Transform Functions[1]	`libmkl_cdft_core.a` `libmkl_core.a`	`libmkl_cdft_core.so` `libmkl_core.so`	`libmkl_cdft_core.a` `libmkl_core.a`	`libmkl_cdft_core.so` `libmkl_core.so`

[1] Add also the library with BLACS routines corresponding to the used MPI. For details, see Linking with ScaLAPACK and Cluster FFTs.

[2] Not applicable.

See Also

- Linking with ScaLAPACK and Cluster FFTs
- Using the Web-based Linking Advisor

Linking with Compiler Support RTLs

You are strongly encouraged to dynamically link in the compatibility OpenMP* run-time library `libiomp`. Link with `libiomp` dynamically even if other libraries are linked statically.

Linking to static OpenMP* run-time library is not recommended because it is very easy with complex software to link in more than one copy of the library. This causes performance problems (too many threads) and may cause correctness problems if more than one copy is initialized.

If you link with dynamic version of `libiomp`, make sure the `LD_LIBRARY_PATH` environment variable is defined correctly.

See Also

- Scripts to Set Environment Variables

Linking with System Libraries

To use the Intel(R) Math Kernel Library (Intel(R) MKL) FFT, Trigonometric Transform, or Poisson, Laplace, and Helmholtz Solver routines, link in the math support system library by adding " `-lm` " to the link line.

On Linux OS, the `libiomp` library relies on the native `pthread` library for multi-threading. Any time `libiomp` is required, add `-lpthread` to your link line afterwards (the order of listing libraries is important).

Linking Examples

See Also

- Using the Web-based Linking Advisor
- Examples for Linking with ScaLAPACK and Cluster FFT

Linking on IA-32 Architecture Systems

The following examples illustrate linking that uses Intel(R) compilers.

The examples use the `.f` Fortran source file. C/C++ users should instead specify a `.cpp` (C++) or `.c` (C) file and replace the `ifort` linker with `icc`

NOTE. If you successfully completed the Setting Environment Variables step of the Getting Started process, you can omit `-I$MKLINCLUDE` in all the examples and omit `-L$MKLPATH` in the examples for dynamic linking.

In these examples,
`MKLPATH=$MKLROOT/lib/ia32`,
`MKLINCLUDE=$MKLROOT/include`:

- Static linking of `myprog.f` and parallel Intel(R) Math Kernel Library (Intel(R) MKL):

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5
-lpthread
```

- Dynamic linking of `myprog.f` and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```

- Static linking of `myprog.f` and sequential version of Intel MKL:

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

```
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_sequential.a
$MKLPATH/libmkl_core.a
-Wl,--end-group
-lpthread
```

- Dynamic linking of `myprog.f` and sequential version of Intel MKL:

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

- Dynamic linking of user code `myprog.f` and parallel or sequential Intel MKL (Call the `mkl_set_threading_layer` function or set value of the `MKL_SET_THREADING_LAYER` environment variable to choose threaded or sequential mode):

```
ifort myprog.f -lmkl_rt
```

- Static linking of `myprog.f`, Fortran 95 LAPACK interface, and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/ia32
-lmkl_lapack95
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a
-Wl,--end-group
-liomp5 -lpthread
```

- Static linking of `myprog.f`, Fortran 95 BLAS interface, and parallel Intel MKL:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/ia32
-lmkl_blas95
-Wl,--start-group $MKLPATH/libmkl_intel.a $MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a
-Wl,--end-group -liomp5 -lpthread
```

See Also

- Fortran 95 Interfaces to LAPACK and BLAS
- Examples for linking a C application using cluster components
- Examples for linking a Fortran application using cluster components
- Using the Single Dynamic Library Interface

Linking on Intel(R) 64 Architecture Systems

The following examples illustrate linking that uses Intel(R) compilers.

The examples use the `.f` Fortran source file. C/C++ users should instead specify a `.cpp` (C++) or `.c` (C) file and replace the `ifort` linker with `icc`

NOTE. If you successfully completed the Setting Environment Variables step of the Getting Started process, you can omit `-I$MKLINCLUDE` in all the examples and omit `-L$MKLPATH` in the examples for dynamic linking.

In these examples,
`MKLPATH=$MKLROOT/lib/intel64`,
`MKLINCLUDE=$MKLROOT/include`:

- Static linking of myprog.f and parallel Intel(R) Math Kernel Library (Intel(R) MKL) supporting the LP64 interface:

```
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 -liomp5 -lpthread
```

- Dynamic linking of myprog.f and parallel Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core
-liomp5 -lpthread
```

- Static linking of myprog.f and sequential version of Intel MKL supporting the LP64 interface:

```
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
```

- Dynamic linking of myprog.f and sequential version of Intel MKL supporting the LP64 interface:

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

- Static linking of myprog.f and parallel Intel MKL supporting the ILP64 interface:

```
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 -liomp5 -lpthread
```

- Dynamic linking of myprog.f and parallel Intel MKL supporting the ILP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE
-lmkl_intel_ilp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```

- Dynamic linking of user code myprog.f and parallel or sequential Intel MKL (Call appropriate functions or set environment variables to choose threaded or sequential mode and to set the interface):

```
ifort myprog.f -lmkl_rt
```

- Static linking of myprog.f, Fortran 95 LAPACK interface, and parallel Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/intel64/lp64
-lmkl_lapack95_lp64 -Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5 -lpthread
```

- Static linking of myprog.f, Fortran 95 BLAS interface, and parallel Intel MKL supporting the LP64 interface:

```
ifort myprog.f -L$MKLPATH -I$MKLINCLUDE -I$MKLINCLUDE/intel64/lp64
-lmkl_blas95_lp64 -Wl,--start-group $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_intel_thread.a
$MKLPATH/libmkl_core.a -Wl,--end-group -liomp5 -lpthread
```

See Also

- Fortran 95 Interfaces to LAPACK and BLAS
- Examples for linking a C application using cluster components
- Examples for linking a Fortran application using cluster components
- Using the Single Dynamic Library Interface

Building Custom Shared Objects

Custom shared objects reduce the collection of functions available in Intel MKL libraries to those required to solve your particular problems, which helps to save disk space and build your own dynamic libraries for distribution.

The Intel(R) Math Kernel Library (Intel(R) MKL) custom shared object builder enables you to create a dynamic library (shared object) containing the selected functions and located in the `tools/builder` directory. The builder contains a makefile and a definition file with the list of functions.

NOTE. The objects in Intel MKL static libraries are position-independent code (PIC), which is not typical for static libraries. Therefore, the custom shared object builder can create a shared object from a subset of Intel MKL functions by picking the respective object files from the static libraries.

Using the Custom Shared Object Builder

To build a custom shared object, use the following command:

`make target [<`*`options`*`>]`

The following table lists possible values of `target` and explains what the command does for each value:

Value	Comment
`libia32`	The command builds static libraries for processors that use the IA-32 architecture. The builder uses the interface, threading, and computational libraries.
`libintel64`	The command builds static libraries for processors that use the Intel(R) 64 architecture. The builder uses the interface, threading, and computational libraries.
`soia32`	The command builds dynamic libraries for processors that use the IA-32 architecture. The builder uses the Single Dynamic Library (SDL) interface library `libmkl_rt.so`.
`sointel64`	The command builds dynamic libraries for processors that use the Intel(R) 64 architecture. The builder uses the SDL interface library `libmkl_rt.so`.
`help`	The command prints Help on the custom shared object builder

The <*`options`*> placeholder stands for the list of parameters that define macros to be used by the makefile. The following table describes these parameters:

Parameter [Values]	Description
`interface = {lp64|ilp64}`	Defines whether to use LP64 or ILP64 programming interface for the Intel 64 architecture. The default value is `lp64`.
`threading = {parallel|sequential}`	Defines whether to use the Intel(R) Math Kernel Library (Intel(R) MKL) in the threaded or sequential mode. The default value is `parallel`.
`export = <`*`file name`*`>`	Specifies the full name of the file that contains the list of entry-point functions to be included in the shared object. The default name is user_list (no extension).

Parameter [Values]	Description
`name = <so name>`	Specifies the name of the library to be created. By default, the names of the created library is `mkl_custom.so`.
`xerbla = <error handler>`	Specifies the name of the object file `<user_xerbla>.o` that contains the user's error handler. The makefile adds this error handler to the library for use instead of the default Intel MKL error handler `xerbla`. If you omit this parameter, the native Intel MKL `xerbla` is used. See the description of the `xerbla` function in the Intel MKL Reference Manual on how to develop your own error handler.
`MKLROOT = <mkl directory>`	Specifies the location of Intel MKL libraries used to build the custom shared object. By default, the builder uses the Intel MKL installation directory.

All the above parameters are optional.

In the simplest case, the command line is `make ia32`, and the missing options have default values. This command creates the `mkl_custom.so` library for processors using the IA-32 architecture. The command takes the list of functions from the `user_list` file and uses the native Intel MKL error handler `xerbla`.

An example of a more complex case follows:

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

In this case, the command creates the `mkl_small.so` library for processors using the IA-32 architecture. The command takes the list of functions from `my_func_list.txt` file and uses the user's error handler `my_xerbla.o`.

The process is similar for processors using the Intel(R) 64 architecture.

See Also

- Using the Single Dynamic Library Interface

Specifying a List of Functions

In the list of functions provided in the `user_list` file, adjust function names to the required interface. For example, for Fortran functions append an underscore character "_" to the names as a suffix:

```
dgemm_
ddot_
dgetrf_
```

If selected functions have several processor-specific versions, they all will be automatically included in the custom library and managed by the dispatcher.

See the `<mkl directory>/tools/builder` folder for complete function domain-specific lists of functions in different function domains.

Distributing Your Custom Shared Object

To enable use of your custom shared object in a threaded mode, distribute `libiomp5.so` along with the custom shared object.

Managing Performance and Memory

5

Optimization Notice

The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

Using Parallelism of the Intel(R) Math Kernel Library

The Intel(R) Math Kernel Library (Intel(R) MKL) is extensively parallelized. See Threaded Functions and Problems for lists of threaded functions and problems that can be threaded.

Intel MKL is *thread-safe*, which means that all Intel MKL functions (except the LAPACK deprecated routine `?lacon`) work correctly during simultaneous execution by multiple threads. In particular, any chunk of threaded Intel MKL code provides access for multiple threads to the same shared data, while permitting only one thread at any given time to access a shared piece of data. Therefore, you can call Intel MKL from multiple threads and not worry about the function instances interfering with each other.

The library uses OpenMP* threading software, so you can use the environment variable `OMP_NUM_THREADS` to specify the number of threads or the equivalent OpenMP run-time function calls. Intel MKL also offers variables that are independent of OpenMP, such as `MKL_NUM_THREADS`, and equivalent Intel MKL functions for thread management. The Intel MKL variables are always inspected first, then the OpenMP variables are examined, and if neither is used, the OpenMP software chooses the default number of threads.

By default, Intel MKL uses the number of threads equal to the number of physical cores on the system.

To achieve higher performance, set the number of threads to the number of real processors or physical cores, as summarized in Techniques to Set the Number of Threads.

See Also

- Managing Multi-core Performance

Threaded Functions and Problems

The following Intel(R) Math Kernel Library (Intel(R) MKL) function domains are threaded:

- Direct sparse solver.
- LAPACK.

  For the list of threaded routines, see Threaded LAPACK Routines.
- Level1 and Level2 BLAS.

  For the list of threaded routines, see Threaded BLAS Level1 and Level2 Routines.
- All Level 3 BLAS and all Sparse BLAS routines except Level 2 Sparse Triangular solvers.
- All mathematical VML functions.
- FFT.

  For the list of FFT transforms that can be threaded, see Threaded FFT Problems.

Threaded LAPACK Routines

In the following list, ? stands for a precision prefix of *each* flavor of the respective routine and may have the value of `s`, `d`, `c`, or `z`.

The following LAPACK routines are threaded:

- Linear equations, computational routines:
  - Factorization: `?getrf, ?gbtrf, ?potrf, ?pptrf, ?sytrf, ?hetrf, ?sptrf, ?hptrf`
  - Solving: `?gbtrs, ?gttrs, ?pptrs, ?pbtrs, ?pttrs, ?sytrs, ?sptrs, ?hptrs, ?tptrs, ?tbtrs`
- Orthogonal factorization, computational routines:
  `?geqrf, ?ormqr, ?unmqr, ?ormlq, ?unmlq, ?ormql, ?unmql, ?ormrq, ?unmrq`
- Singular Value Decomposition, computational routines:
  `?gebrd, ?bdsqr`
- Symmetric Eigenvalue Problems, computational routines:
  `?sytrd, ?hetrd, ?sptrd, ?hptrd, ?steqr, ?stedc`.
- Generalized Nonsymmetric Eigenvalue Problems, computational routines:
  `chgeqz/zhgeqz`.

A number of other LAPACK routines, which are based on threaded LAPACK or BLAS routines, make effective use of parallelism:
`?gesv, ?posv, ?gels, ?gesvd, ?syev, ?heev, cgegs/zgegs, cgegv/zgegv, cgges/zgges, cggesx/zggesx, cggev/zggev, cggevx/zggevx`, and so on.

Threaded BLAS Level1 and Level2 Routines

In the following list, ? stands for a precision prefix of *each* flavor of the respective routine and may have the value of `s`, `d`, `c`, or `z`.

The following routines are threaded for Intel(R) Core™2 Duo and Intel(R) Core™ i7 processors:

- Level1 BLAS:

`?axpy, ?copy, ?swap, ddot/sdot, cdotc, drot/srot`

- Level2 BLAS:
  `?gemv, ?trmv, dsyr/ssyr, dsyr2/ssyr2, dsymv/ssymv`

Threaded FFT Problems

The following characteristics of a specific problem determine whether your FFT computation may be threaded:

- rank
- domain
- size/length
- precision (single or double)
- placement (in-place or out-of-place)
- strides
- number of transforms
- layout (for example, interleaved or split layout of complex data)

Most FFT problems are threaded. In particular, computation of multiple transforms in one call (number of transforms > 1) is threaded. Details of which transforms are threaded follow.

One-dimensional (1D) transforms

1D transforms are threaded in many cases.

1D complex-to-complex (c2c) transforms of size *N* using interleaved complex data layout are threaded under the following conditions depending on the architecture:

Architecture	Conditions
Intel(R) 64	*N* is a power of 2, $log_2(N) > 9$, the transform is double-precision out-of-place, and input/output strides equal 1.
IA-32	*N* is a power of 2, $log_2(N) > 13$, and the transform is single-precision.
	N is a power of 2, $log_2(N) > 14$, and the transform is double-precision.
Any	*N* is composite, $log_2(N) > 16$, and input/output strides equal 1.

1D real-to-complex and complex-to-real transforms are not threaded.

1D complex-to-complex transforms using split-complex layout are not threaded.

Prime-size complex-to-complex 1D transforms are not threaded.

Multidimensional transforms

All multidimensional transforms on large-volume data are threaded.

Avoiding Conflicts in the Execution Environment

Certain situations can cause conflicts in the execution environment that make the use of threads in Intel(R) Math Kernel Library (Intel(R) MKL) problematic. This section briefly discusses why these problems exist and how to avoid them.

If you thread the program using OpenMP directives and compile the program with Intel(R) compilers, Intel MKL and the program will both use the same threading library. Intel MKL tries to determine if it is in a parallel region in the program, and if it is, it does not spread its operations over multiple threads unless you specifically request Intel MKL to do so via the `MKL_DYNAMIC` functionality. However, Intel MKL can be aware that it is in a parallel region only if the threaded program and Intel MKL are using the same threading library. If your program is threaded by some other means, Intel MKL may operate in multithreaded mode, and the performance may suffer due to overuse of the resources.

The following table considers several cases where the conflicts may arise and provides recommendations depending on your threading model:

Threading model	Discussion
You thread the program using OS threads (`pthreads` on Linux* OS).	If more than one thread calls Intel MKL, and the function being called is threaded, it may be important that you turn off Intel MKL threading. Set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads).
You thread the program using OpenMP directives and/or pragmas and compile the program using a compiler other than a compiler from Intel.	This is more problematic because setting of the `OMP_NUM_THREADS` environment variable affects both the compiler's threading library and `libiomp`. In this case, choose the threading library that matches the layered Intel MKL with the OpenMP compiler you employ (see Linking Examples on how to do this). If this is not possible, use Intel MKL in the sequential mode. To do this, you should link with the appropriate threading library: `libmkl_sequential.a` or `libmkl_sequential.so` (see High-level Directory Structure).
There are multiple programs running on a multiple-cpu system, for example, a parallelized program that runs using MPI for communication in which each processor is treated as a node.	The threading software will see multiple processors on the system even though each processor has a separate MPI process running on it. In this case, one of the solutions is to set the number of threads to one by any of the available means (see Techniques to Set the Number of Threads). Section Intel(R) Optimized MP LINPACK Benchmark for Clusters discusses another solution for a Hybrid (OpenMP* + MPI) mode.

See Also

- Using Additional Threading Control
- Linking with Compiler Support RTLs

Techniques to Set the Number of Threads

Use one of the following techniques to change the number of threads to use in the Intel(R) Math Kernel Library (Intel(R) MKL):

- Set one of the OpenMP or Intel MKL environment variables:
  - `OMP_NUM_THREADS`
  - `MKL_NUM_THREADS`
  - `MKL_DOMAIN_NUM_THREADS`
- Call one of the OpenMP or Intel MKL functions:
  - `omp_set_num_threads()`

- `mkl_set_num_threads()`
- `mkl_domain_set_num_threads()`

When choosing the appropriate technique, take into account the following rules:

- The Intel MKL threading controls take precedence over the OpenMP controls because they are inspected first.
- A function call takes precedence over any environment variables. The exception, which is a consequence of the previous rule, is the OpenMP subroutine `omp_set_num_threads()`, which does not have precedence over Intel MKL environment variables, such as `MKL_NUM_THREADS`. See Using Additional Threading Control for more details.
- You cannot change run-time behavior in the course of the run using the environment variables because they are read only once at the first call to Intel MKL.

Setting the Number of Threads Using an OpenMP* Environment Variable

You can set the number of threads using the environment variable `OMP_NUM_THREADS`. To change the number of threads, use the appropriate command in the command shell in which the program is going to run, for example:

- For the bash shell, enter:
  `export OMP_NUM_THREADS=<number of threads to use>`
- For the csh or tcsh shell, enter:
  `set OMP_NUM_THREADS=<number of threads to use>`

See Also

- Using Additional Threading Control

Changing the Number of Threads at Run Time

You cannot change the number of threads during run time using environment variables. However, you can call OpenMP API functions from your program to change the number of threads during run time. The following sample code shows how to change the number of threads during run time using the `omp_set_num_threads()` routine. See also Techniques to Set the Number of Threads.

The following example shows both C and Fortran code examples. To run this example in the C language, use the `omp.h` header file from the Intel(R) compiler package. If you do not have the Intel compiler but wish to explore the functionality in the example, use Fortran API for `omp_set_num_threads()` rather than the C version. For example, `omp_set_num_threads_( &i_one );`

```
// ******* C language *******
#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);
  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;
   }
  }
  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;
  }


// ******* Fortran language *******
  PROGRAM DGEMM_DIFF_THREADS
  INTEGER N, I, J
  PARAMETER (N=1000)
  REAL*8 A(N,N),B(N,N),C(N,N)
  REAL*8 ALPHA, BETA
  INTEGER*8 MKL_MALLOC
  integer ALLOC_SIZE
  integer NTHRS
  ALLOC_SIZE = 8*N*N
  A_PTR = MKL_MALLOC(ALLOC_SIZE,128)
  B_PTR = MKL_MALLOC(ALLOC_SIZE,128)
  C_PTR = MKL_MALLOC(ALLOC_SIZE,128)
  ALPHA = 1.1
  BETA = -1.2
```

```
DO I=1,N
DO J=1,N
 A(I,J) = I+J
 B(I,J) = I*j
 C(I,J) = 0.0
END DO
END DO
CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)
print *,'Row A C'
DO i=1,10
write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
END DO
CALL OMP_SET_NUM_THREADS(1);
DO I=1,N
DO J=1,N
 A(I,J) = I+J
 B(I,J) = I*j
 C(I,J) = 0.0
END DO
END DO
CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)
print *,'Row A C'
DO i=1,10
write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
END DO
CALL OMP_SET_NUM_THREADS(2);
DO I=1,N
DO J=1,N
 A(I,J) = I+J
 B(I,J) = I*j
 C(I,J) = 0.0
END DO
END DO
CALL DGEMM('N','N',N,N,N,ALPHA,A,N,B,N,BETA,C,N)
print *,'Row A C'
DO i=1,10
write(*,'(I4,F20.8,F20.8)') I, A(1,I),C(1,I)
END DO
STOP
END
```

Using Additional Threading Control

Intel(R) MKL-specific Environment Variables for Threading Control

The Intel(R) Math Kernel Library (Intel(R) MKL) provides optional threading controls, that is, the environment variables and service functions that are independent of OpenMP. They behave similar to their OpenMP equivalents, but take precedence over them in the meaning that the MKL-specific threading controls are inspected first. By using these controls along with OpenMP variables, you can thread the part of the application that does not call Intel MKL and the library independently from each other.

These controls enable you to specify the number of threads for Intel MKL independently of the OpenMP settings. Although Intel MKL may actually use a different number of threads from the number suggested, the controls will also enable you to instruct the library to try using the suggested number when the number used in the calling application is unavailable.

NOTE. Sometimes Intel MKL does not have a choice on the number of threads for certain reasons, such as system resources.

Use of the Intel MKL threading controls in your application is optional. If you do not use them, the library will mainly behave the same way as Intel MKL 9.1 in what relates to threading with the possible exception of a different default number of threads.

Section "Number of User Threads" in the "Fourier Transform Functions" chapter of the *Intel MKL Reference Manual* shows how the Intel MKL threading controls help to set the number of threads for the FFT computation.

The table below lists the Intel MKL environment variables for threading control, their equivalent functions, and OMP counterparts:

Environment Variable	**Service Function**	**Comment**	**Equivalent OpenMP* Environment Variable**
`MKL_NUM_THREADS`	`mkl_set_num_threads`	Suggests the number of threads to use.	`OMP_NUM_THREADS`
`MKL_DOMAIN_NUM_THREADS`	`mkl_domain_set_num_threads`	Suggests the number of threads for a particular function domain.	
`MKL_DYNAMIC`	`mkl_set_dynamic`	Enables Intel MKL to dynamically change the number of threads.	`OMP_DYNAMIC`

NOTE. The functions take precedence over the respective environment variables. Therefore, if you want Intel MKL to use a given number of threads in your application and do not want users of your application to change this number using environment variables, set the number of threads by a call to `mkl_set_num_threads()`, which will have full precedence over any environment variables being set.

The example below illustrates the use of the Intel MKL function `mkl_set_num_threads()` to set one thread.

```
// ******* C language *******
#include <omp.h>
#include <mkl.h>
...
mkl_set_num_threads ( 1 );


// ******* Fortran language *******
...
call mkl_set_num_threads( 1 )
```

See the *Intel MKL Reference Manual* for the detailed description of the threading control functions, their parameters, calling syntax, and more code examples.

MKL_DYNAMIC

The MKL_DYNAMIC environment variable enables the Intel(R) Math Kernel Library (Intel(R) MKL) to dynamically change the number of threads.

The default value of MKL_DYNAMIC is TRUE, regardless of OMP_DYNAMIC, whose default value may be FALSE.

When MKL_DYNAMIC is TRUE, Intel MKL tries to use what it considers the best number of threads, up to the maximum number you specify.

For example, MKL_DYNAMIC set to TRUE enables optimal choice of the number of threads in the following cases:

- If the requested number of threads exceeds the number of physical cores (perhaps because of hyper-threading), and MKL_DYNAMIC is not changed from its default value of TRUE, Intel MKL will scale down the number of threads to the number of physical cores.
- If you are able to detect the presence of MPI, but cannot determine if it has been called in a thread-safe mode (it is impossible to detect this with MPICH 1.2.x, for instance), and MKL_DYNAMIC has not been changed from its default value of TRUE, Intel MKL will run one thread.

When MKL_DYNAMIC is FALSE, Intel MKL tries not to deviate from the number of threads the user requested. However, setting MKL_DYNAMIC=FALSE does not ensure that Intel MKL will use the number of threads that you request. The library may have no choice on this number for such reasons as system resources. Additionally, the library may examine the problem and use a different number of threads than the value suggested. For example, if you attempt to do a size one matrix-matrix multiply across eight threads, the library may instead choose to use only one thread because it is impractical to use eight threads in this event.

Note also that if Intel MKL is called in a parallel region, it will use only one thread by default. If you want the library to use nested parallelism, and the thread within a parallel region is compiled with the same OpenMP compiler as Intel MKL is using, you may experiment with setting MKL_DYNAMIC to FALSE and manually increasing the number of threads.

In general, set MKL_DYNAMIC to FALSE only under circumstances that Intel MKL is unable to detect, for example, to use nested parallelism where the library is already called from a parallel section.

MKL_DOMAIN_NUM_THREADS

The MKL_DOMAIN_NUM_THREADS environment variable suggests the number of threads for a particular function domain.

MKL_DOMAIN_NUM_THREADS accepts a string value <MKL-env-string>, which must have the following format:

<MKL-env-string> ::= *<MKL-domain-env-string>* { *<delimiter>* *<MKL-domain-env-string>* }

<delimiter> ::= [*<space-symbol>**] (*<space-symbol>* | *<comma-symbol>* | *<semicolon-symbol>* | *<colon-symbol>*) [*<space-symbol>**]

<MKL-domain-env-string> ::= *<MKL-domain-env-name>* *<uses>* *<number-of-threads>*

<MKL-domain-env-name> ::= MKL_ALL | MKL_BLAS | MKL_FFT | MKL_VML

<uses> ::= [*<space-symbol>**] (*<space-symbol>* | *<equality-sign>* | *<comma-symbol>*) [*<space-symbol>**]

<number-of-threads> ::= *<positive-number>*

<positive-number> ::= *<decimal-positive-number>* | *<octal-number>* | *<hexadecimal-number>*

In the syntax above, MKL_BLAS indicates the BLAS function domain, MKL_FFT indicates non-cluster FFTs, and MKL_VML indicates the Vector Mathematics Library.

For example,

```
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 .
```

The global variables MKL_ALL, MKL_BLAS, MKL_FFT, and MKL_VML, as well as the interface for the Intel(R) Math Kernel Library (Intel(R) MKL) threading control functions, can be found in the mkl.h header file.

The table below illustrates how values of MKL_DOMAIN_NUM_THREADS are interpreted.

Value of MKL_DOMAIN_NUM_THREADS	**Interpretation**
MKL_ALL=4	All parts of Intel MKL should try four threads. The actual number of threads may be still different because of the MKL_DYNAMIC setting or system resource issues. The setting is equivalent to MKL_NUM_THREADS = 4.
MKL_ALL=1, MKL_BLAS=4	All parts of Intel MKL should try one thread, except for BLAS, which is suggested to try four threads.
MKL_VML=2	VML should try two threads. The setting affects no other part of Intel MKL.

Be aware that the domain-specific settings take precedence over the overall ones. For example, the "MKL_BLAS=4" value of MKL_DOMAIN_NUM_THREADS suggests trying four threads for BLAS, regardless of later setting MKL_NUM_THREADS, and a function call "mkl_domain_set_num_threads (4, MKL_BLAS);" suggests the same, regardless of later calls to mkl_set_num_threads().
However, a function call with input "MKL_ALL", such as "mkl_domain_set_num_threads (4, MKL_ALL);" is equivalent to "mkl_set_num_threads(4)", and thus it will be overwritten by later calls to mkl_set_num_threads. Similarly, the environment setting of MKL_DOMAIN_NUM_THREADS with "MKL_ALL=4" will be overwritten with MKL_NUM_THREADS = 2.

Whereas the MKL_DOMAIN_NUM_THREADS environment variable enables you set several variables at once, for example, "MKL_BLAS=4,MKL_FFT=2", the corresponding function does not take string syntax. So, to do the same with the function calls, you may need to make several calls, which in this example are as follows:

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

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

Setting the Environment Variables for Threading Control

To set the environment variables used for threading control, in the command shell in which the program is going to run, enter the export or set commands, depending on the shell you use. For example, for a bash shell, use the export commands:

```
export <VARIABLE NAME>=<value>
```

For example:

```
export MKL_NUM_THREADS=4
```

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

For the csh or tcsh shell, use the `set` commands.

`set` *<VARIABLE NAME>=<value>*.

For example:

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

Tips and Techniques to Improve Performance

Coding Techniques

To obtain the best performance with the Intel(R) Math Kernel Library (Intel(R) MKL), ensure the following data alignment in your source code:

- Align arrays on 16-byte boundaries.
- Make sure leading dimension values (`n*element_size`) of two-dimensional arrays are divisible by 16.
- For two-dimensional arrays, avoid leading dimension values divisible by 2048.

LAPACK Packed Routines

The routines with the names that contain the letters `HP, OP, PP, SP, TP, UP` in the matrix type and storage position (the second and third letters respectively) operate on the matrices in the packed format (see LAPACK "Routine Naming Conventions" sections in the Intel MKL Reference Manual). Their functionality is strictly equivalent to the functionality of the unpacked routines with the names containing the letters `HE, OR, PO, SY, TR, UN` in the same positions, but the performance is significantly lower.

If the memory restriction is not too tight, use an unpacked routine for better performance. In this case, you need to allocate $N^2/2$ more memory than the memory required by a respective packed routine, where N is the problem size (the number of equations).

For example, to speed up solving a symmetric eigenproblem with an expert driver, use the unpacked routine:

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

where a is the dimension *lda*-by-*n*, which is at least N^2 elements,
instead of the packed routine:

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

where ap is the dimension $N*(N+1)/2$.

FFT Functions

Additional conditions can improve performance of the FFT functions.

The addresses of the first elements of arrays and the leading dimension values, in bytes (`n*element_size`), of two-dimensional arrays should be divisible by cache line size, which equals:

- 32 bytes for the Intel(R) Pentium(R) III processors
- 64 bytes for the Intel(R) Pentium(R) 4 processors and processors using Intel(R) 64 architecture

Hardware Configuration Tips

Dual-Core Intel(R) Xeon(R) processor 5100 series systems

To get the best performance with the Intel(R) Math Kernel Library (Intel(R) MKL) on Dual-Core Intel(R) Xeon(R) processor 5100 series systems, enable the Hardware DPL (streaming data) Prefetcher functionality of this processor. To configure this functionality, use the appropriate BIOS settings, as described in your BIOS documentation.

The use of Hyper-Threading Technology

Hyper-Threading Technology (HT Technology) is especially effective when each thread performs different types of operations and when there are under-utilized resources on the processor. However, Intel MKL fits neither of these criteria because the threaded portions of the library execute at high efficiencies using most of the available resources and perform identical operations on each thread. You may obtain higher performance by disabling HT Technology.

If you run with HT enabled, performance may be especially impacted if you run on fewer threads than physical cores. Moreover, if, for example, there are two threads to every physical core, the thread scheduler may assign two threads to some cores and ignore the other cores altogether. If you are using the OpenMP* library of the Intel Compiler, read the respective User Guide on how to best set the thread affinity interface to avoid this situation. For Intel MKL, apply the following setting:

```
set KMP_AFFINITY=granularity=fine,compact,1,0
```

See Also

- Using Parallelism of the Intel(R) Math Kernel Library

Managing Multi-core Performance

You can obtain best performance on systems with multi-core processors by requiring that threads do not migrate from core to core. To do this, bind threads to the CPU cores by setting an affinity mask to threads. Use one of the following options:

- OpenMP facilities (recommended, if available), for example, the `KMP_AFFINITY` environment variable using the Intel OpenMP library
- A system function, as explained below

Consider the following performance issue:

- The system has two sockets with two cores each, for a total of four cores (CPUs)
- The two -thread parallel application that calls the Intel(R) Math Kernel Library (Intel(R) MKL) FFT happens to run faster than in four threads, but the performance in two threads is very unstable

The following code example shows how to resolve this issue by setting an affinity mask by operating system means using the Intel(R) compiler. The code calls the system function `sched_setaffinity` to bind the threads to the cores on different sockets. Then the Intel MKL FFT function is called:

```
#define _GNU_SOURCE //for using the GNU CPU affinity
// (works with the appropriate kernel and glibc)
// Set affinity mask
#include <sched.h>
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main(void) {
 int NCPUs = sysconf(_SC_NPROCESSORS_CONF);
 printf("Using thread affinity on %i NCPUs\n", NCPUs);
#pragma omp parallel default(shared)
 {
  cpu_set_t new_mask;
  cpu_set_t was_mask;
  int tid = omp_get_thread_num();

  CPU_ZERO(&new_mask);

  // 2 packages x 2 cores/pkg x 1 threads/core (4 total cores)
  CPU_SET(tid==0 ? 0 : 2, &new_mask);

  if (sched_getaffinity(0, sizeof(was_mask), &was_mask) == -1) {
   printf("Error: sched_getaffinity(%d, sizeof(was_mask), &was_mask)\n", tid);
  }
  if (sched_setaffinity(0, sizeof(new_mask), &new_mask) == -1) {
   printf("Error: sched_setaffinity(%d, sizeof(new_mask), &new_mask)\n", tid);
  }
  printf("tid=%d new_mask=%08X was_mask=%08X\n", tid,
      *(unsigned int*)(&new_mask), *(unsigned int*)(&was_mask));
 }
 // Call Intel MKL FFT function
 return 0;
}
```

Compile the application with the Intel compiler using the following command:

```
icc test_application.c -openmp
```

where `test_application.c` is the filename for the application.

Build the application. Run it in two threads, for example, by using the environment variable to set the number of threads:

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

See the *Linux Programmer's Manual* (in man pages format) for particulars of the `sched_setaffinity` function used in the above example.

Operating on Denormals

The IEEE 754-2008 standard, "An IEEE Standard for Binary Floating-Point Arithmetic", defines *denormal* (or *subnormal*) numbers as non-zero numbers smaller than the smallest possible normalized numbers for a specific floating-point format. Floating-point operations on denormals are slower than on normalized operands because denormal operands and results are usually handled through a software assist mechanism rather than directly in hardware. This software processing causes Intel(R) Math Kernel Library (Intel(R) MKL) functions that consume denormals to run slower than with normalized floating-point numbers.

You can mitigate this performance issue by setting the appropriate bit fields in the MXCSR floating-point control register to flush denormals to zero (FTZ) or to replace any denormals loaded from memory with zero (DAZ). Check your compiler documentation to determine whether it has options to control FTZ and DAZ. Note that these compiler options may slightly affect accuracy.

FFT Optimized Radices

You can improve the performance of the Intel(R) Math Kernel Library (Intel(R) MKL) FFT if the length of your data vector permits factorization into powers of optimized radices.

In Intel MKL, the optimized radices are 2, 3, 5, 7, and 11.

Using Memory Management

Memory Management Software of the Intel(R) Math Kernel Library

Intel(R) Math Kernel Library (Intel(R) MKL) has memory management software that controls memory buffers for the use by the library functions. New buffers that the library allocates when your application calls Intel MKL are not deallocated until the program ends. To get the amount of memory allocated by the memory management software, call the `mkl_mem_stat()` function. If your program needs to free memory, call `mkl_free_buffers()`. If another call is made to a library function that needs a memory buffer, the memory manager again allocates the buffers and they again remain allocated until either the program ends or the program deallocates the memory. This behavior facilitates better performance. However, some tools may report this behavior as a memory leak.

The memory management software is turned on by default, which leaves memory allocated by calls to Intel MKL until the program ends. To disable this behavior of the memory management software, set the `MKL_DISABLE_FAST_MM` environment variable to any value. This configures the memory management software to allocate and free memory from call to call. Disabling this feature will negatively impact performance of some Intel MKL routines, especially for small problem sizes.

Redefining Memory Functions

In C/C++ programs, you can replace Intel(R) Math Kernel Library (Intel(R) MKL) memory functions that the library uses by default with their own functions. To do this, use the *memory renaming* feature.

Memory Renaming

Intel MKL memory management by default uses standard C run-time memory functions to allocate or free memory. These functions can be replaced using memory renaming.

Intel MKL accesses the memory functions by pointers `i_malloc, i_free, i_calloc`, and `i_realloc`, which are visible at the application level. These pointers initially hold addresses of the standard C run-time memory functions `malloc, free, calloc`, and `realloc`, respectively. You can programmatically redefine values of these pointers to the addresses of your application's memory management functions.

Redirecting the pointers is the only correct way to use your own set of memory management functions. If you call your own memory functions without redirecting the pointers, the memory will get managed by two independent memory management packages, which may cause unexpected memory issues.

How to Redefine Memory Functions

To redefine memory functions, use the following procedure:

1. Include the `i_malloc.h` header file in your code.
   This header file contains all declarations required for replacing the memory allocation functions. The header file also describes how memory allocation can be replaced in those Intel libraries that support this feature.
2. Redefine values of pointers `i_malloc, i_free, i_calloc`, and `i_realloc` prior to the first call to MKL functions, as shown in the following example:

```
#include "i_malloc.h"
  . . .
  i_malloc  = my_malloc;
  i_calloc  = my_calloc;
  i_realloc = my_realloc;
  i_free    = my_free;
  . . .
// Now you may call Intel MKL functions
```

Language-specific Usage Options 6

The Intel(R) Math Kernel Library (Intel(R) MKL) provides broad support for Fortran and C/C++ programming. However, not all function domains support both Fortran and C interfaces. For example, LAPACK has no C interface. You can call functions comprising such domains from C using mixed-language programming.

If you want to use LAPACK or BLAS, which support Fortran, in the Fortran 95 environment, additional effort may be initially required to build compiler-specific interface libraries and modules from the source code provided with Intel MKL.

Optimization Notice
The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors. While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Intel recommends that you evaluate other library products to determine which best meets your requirements.

Using Language-Specific Interfaces with Intel(R) Math Kernel Library

This section discusses mixed-language programming and the use of language-specific interfaces with Intel(R) Math Kernel Library (Intel(R) MKL).

See also *Appendix G in the Intel MKL Reference Manual* for details of the FFTW interfaces to Intel MKL.

Interface Libraries and Modules

You can create the following interface libraries and modules using the respective makefiles located in the interfaces directory.

File name	Contains
Libraries, in Intel(R) Math Kernel Library (Intel(R) MKL) architecture-specific directories	
libmkl_blas95.a[1]	Fortran 95 wrappers for BLAS (BLAS95) for IA-32 architecture.
libmkl_blas95_ilp64.a[1]	Fortran 95 wrappers for BLAS (BLAS95) supporting LP64 interface.
libmkl_blas95_lp64.a[1]	Fortran 95 wrappers for BLAS (BLAS95) supporting ILP64 interface.
libmkl_lapack95.a[1]	Fortran 95 wrappers for LAPACK (LAPACK95) for IA-32 architecture.
libmkl_lapack95_lp64.a[1]	Fortran 95 wrappers for LAPACK (LAPACK95) supporting LP64 interface.
libmkl_lapack95_ilp64.a[1]	Fortran 95 wrappers for LAPACK (LAPACK95) supporting ILP64 interface.
libfftw2xc_intel.a[1]	Interfaces for FFTW version 2.x (C interface for Intel(R) compilers) to call Intel MKL FFTs.
libfftw2xc_gnu.a	Interfaces for FFTW version 2.x (C interface for GNU compilers) to call Intel MKL FFTs.
libfftw2xf_intel.a	Interfaces for FFTW version 2.x (Fortran interface for Intel compilers) to call Intel MKL FFTs.
libfftw2xf_gnu.a	Interfaces for FFTW version 2.x (Fortran interface for GNU compiler) to call Intel MKL FFTs.
libfftw3xc_intel.a[2]	Interfaces for FFTW version 3.x (C interface for Intel compiler) to call Intel MKL FFTs.
libfftw3xc_gnu.a	Interfaces for FFTW version 3.x (C interface for GNU compilers) to call Intel MKL FFTs.
libfftw3xf_intel.a[2]	Interfaces for FFTW version 3.x (Fortran interface for Intel compilers) to call Intel MKL FFTs.
libfftw3xf_gnu.a	Interfaces for FFTW version 3.x (Fortran interface for GNU compilers) to call Intel MKL FFTs.
libfftw2x_cdft_SINGLE.a	Single-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs.
libfftw2x_cdft_DOUBLE.a	Double-precision interfaces for MPI FFTW version 2.x (C interface) to call Intel MKL cluster FFTs.
libfftw3x_cdft.a	Interfaces for MPI FFTW version 3.x (C interface) to call Intel MKL cluster FFTs.

File name	Contains
`libfftw3x_cdft_ilp64.a`	Interfaces for MPI FFTW version 3.x (C interface) to call Intel MKL cluster FFTs supporting the ILP64 interface.
Modules, in architecture- and interface-specific subdirectories of the Intel MKL include directory	
`blas95.mod`[1]	Fortran 95 interface module for BLAS (BLAS95).
`lapack95.mod`[1]	Fortran 95 interface module for LAPACK (LAPACK95).
`f95_precision.mod`[1]	Fortran 95 definition of precision parameters for BLAS95 and LAPACK95.
`mkl95_blas.mod`[1]	Fortran 95 interface module for BLAS (BLAS95), identical to blas95.mod. To be removed in one of the future releases.
`mkl95_lapack.mod`[1]	Fortran 95 interface module for LAPACK (LAPACK95), identical to lapack95.mod. To be removed in one of the future releases.
`mkl95_precision.mod`[1]	Fortran 95 definition of precision parameters for BLAS95 and LAPACK95, identical to `f95_precision.mod`. To be removed in one of the future releases.
`mkl_service.mod`[1]	Fortran 95 interface module for Intel MKL support functions.

[1]Prebuilt for the Intel(R) Fortran compiler

[2]FFTW3 interfaces are integrated with Intel MKL. Look into `<`*`mkl directory`*`>/interfaces/fftw3x*/makefile` for options defining how to build and where to place the standalone library with the wrappers.

See Also

- Fortran 95 Interfaces to LAPACK and BLAS

Fortran 95 Interfaces to LAPACK and BLAS

Fortran 95 interfaces are compiler-dependent. Intel(R) Math Kernel Library (Intel(R) MKL) provides the interface libraries and modules precompiled with the Intel(R) Fortran compiler. Additionally, the Fortran 95 interfaces and wrappers are delivered as sources. (For more information, see Compiler-dependent Functions and Fortran 90 Modules). If you are using a different compiler, build the appropriate library and modules with your compiler and link the library as a user's library:

1. Go to the respective directory `<`*`mkl directory`*`>/interfaces/blas95` or `<`*`mkl directory`*`>/interfaces/lapack95`
2. Type one of the following commands depending on your architecture:
   - For the IA-32 architecture,

     `make libia32 INSTALL_DIR=<`*`user dir`*`>`
   - For the Intel(R) 64 architecture,

     `make libintel64 [interface=lp64|ilp64] INSTALL_DIR=<`*`user dir`*`>`

NOTE. Parameter `INSTALL_DIR` is required.

As a result, the required library is built and installed in the `<user dir>/lib` directory, and the `.mod` files are built and installed in the `<user dir>/include/<arch>[/{lp64|ilp64}]` directory, where `<arch>` is one of `{ia32, intel64}`.

By default, the ifort compiler is assumed. You may change the compiler with an additional parameter of `make`: `FC=<compiler>`.

For example, command

```
make libintel64 FC=pgf95 INSTALL_DIR=<user pgf95 dir> interface=lp64
```

builds the required library and `.mod` files and installs them in subdirectories of `<user pgf95 dir>`.

To delete the library from the building directory, use the following commands:

- For the IA-32 architecture,

  `make cleania32 INSTALL_DIR=<user dir>`
- For the Intel(R) 64 architecture,

  `make cleanintel64 [interface=lp64|ilp64] INSTALL_DIR=<user dir>`
- For all the architectures,

  `make clean INSTALL_DIR=<user dir>`

CAUTION. Even if you have administrative rights, avoid setting `INSTALL_DIR=../..` or `INSTALL_DIR=<mkl directory>` in a build or clean command above because these settings replace or delete the Intel MKL prebuilt Fortran 95 library and modules.

Compiler-dependent Functions and Fortran 90 Modules

Compiler-dependent functions occur whenever the compiler inserts into the object code function calls that are resolved in its run-time library (RTL). Linking of such code without the appropriate RTL will result in undefined symbols. Intel(R) Math Kernel Library (Intel(R) MKL) has been designed to minimize RTL dependencies.

In cases where RTL dependencies might arise, the functions are delivered as source code and you need to compile the code with whatever compiler you are using for your application.

In particular, Fortran 90 modules result in the compiler-specific code generation requiring RTL support. Therefore, Intel MKL delivers these modules compiled with the Intel(R) compiler, along with source code, to be used with different compilers.

Mixed-language Programming with the Intel(R) Math Kernel Library

Appendix A: Intel(R) Math Kernel Library Language Interfaces Support lists the programming languages supported for each Intel(R) Math Kernel Library (Intel(R) MKL) function domain. However, you can call Intel MKL routines from different language environments.

Calling LAPACK, BLAS, and CBLAS Routines from C/C++ Language Environments

Not all Intel(R) Math Kernel Library (Intel(R) MKL) function domains support both C and Fortran environments. To use Intel MKL Fortran-style functions in C/C++ environments, you should observe certain conventions, which are discussed for LAPACK and BLAS in the subsections below.

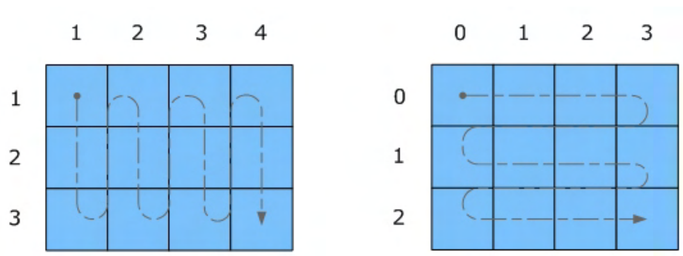
CAUTION. Avoid calling BLAS 95/LAPACK 95 from C/C++. Such calls require skills in manipulating the descriptor of a deferred-shape array, which is the Fortran 90 type. Moreover, BLAS95/LAPACK95 routines contain links to a Fortran RTL.

LAPACK and BLAS

Because LAPACK and BLAS routines are Fortran-style, when calling them from C-language programs, follow the Fortran-style calling conventions:

- Pass variables by *address*, not by *value*.
  Function calls in Example "Calling a Complex BLAS Level 1 Function from C++" and Example "Using CBLAS Interface Instead of Calling BLAS Directly from C" illustrate this.
- Store your data in Fortran style, that is, column-major rather than row-major order.

With row-major order, adopted in C, the last array index changes most quickly and the first one changes most slowly when traversing the memory segment where the array is stored. With Fortran-style column-major order, the last index changes most slowly whereas the first index changes most quickly (as illustrated by the figure below for a two-dimensional array).



A: Column-major order (Fortran-style)   B: Row-major order (C-style)

For example, if a two-dimensional matrix A of size `m x n` is stored densely in a one-dimensional array B, you can access a matrix element like this:

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

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

When calling LAPACK or BLAS routines from C, be aware that because the Fortran language is case-insensitive, the routine names can be both upper-case or lower-case, with or without the trailing underscore. For example, the following names are equivalent:

- LAPACK: `dgetrf`, `DGETRF`, `dgetrf_`, and `DGETRF_`
- BLAS: `dgemm`, `DGEMM`, `dgemm_`, and `DGEMM_`

See Example "Calling a Complex BLAS Level 1 Function from C++" on how to call BLAS routines from C.

See also the Intel(R) MKL Reference Manual for a description of the C interface to LAPACK functions.

CBLAS

Instead of calling BLAS routines from a C-language program, you can use the CBLAS interface.

CBLAS is a C-style interface to the BLAS routines. You can call CBLAS routines using regular C-style calls. Use the `mkl.h` header file with the CBLAS interface. The header file specifies enumerated values and prototypes of all the functions. It also determines whether the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. Example "Using CBLAS Interface Instead of Calling BLAS Directly from C" illustrates the use of the CBLAS interface.

C Interface to LAPACK

Instead of calling LAPACK routines from a C-language program, you can use the C interface to LAPACK provided by Intel MKL.

The C interface to LAPACK is a C-style interface to the LAPACK routines. This interface supports matrices in row-major and column-major order, which you can define in the first function argument *`matrix_order`*. Use the `mkl_lapacke.h` header file with the C interface to LAPACK. The header file specifies constants and prototypes of all the functions. It also determines whether the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation. You can find examples of the C interface to LAPACK in the `examples/lapacke` subdirectory in the Intel MKL installation directory.

Using Complex Types in C/C++

As described in the *Building Applications* document for the Intel(R) Fortran Compiler XE, C/C++ does not directly implement the Fortran types `COMPLEX(4)` and `COMPLEX(8)`. However, you can write equivalent structures. The type `COMPLEX(4)` consists of two 4-byte floating-point numbers. The first of them is the real-number component, and the second one is the imaginary-number component. The type `COMPLEX(8)` is similar to `COMPLEX(4)` except that it contains two 8-byte floating-point numbers.

Intel(R) Math Kernel Library (Intel(R) MKL) provides complex types `MKL_Complex8` and `MKL_Complex16`, which are structures equivalent to the Fortran complex types `COMPLEX(4)` and `COMPLEX(8)`, respectively. The `MKL_Complex8` and `MKL_Complex16` types are defined in the `mkl_types.h` header file. You can use these types to define complex data. You can also redefine the types with your own types before including the `mkl_types.h` header file. The only requirement is that the types must be compatible with the Fortran complex layout, that is, the complex type must be a pair of real numbers for the values of real and imaginary parts.

For example, you can use the following definitions in your C++ code:

```
#define MKL_Complex8 std::complex<float>
```

and

```
#define MKL_Complex16 std::complex<double>
```

See Example "Calling a Complex BLAS Level 1 Function from C++" for details. You can also define these types in the command line:

```
-DMKL_Complex8="std::complex<float>"
-DMKL_Complex16="std::complex<double>"
```

Calling BLAS Functions that Return the Complex Values in C/C++ Code

Complex values that functions return are handled differently in C and Fortran. Because BLAS is Fortran-style, you need to be careful when handling a call from C to a BLAS function that returns complex values. However, in addition to normal function calls, Fortran enables calling functions as though they were subroutines, which provides a mechanism for returning the complex value correctly when the function is called from a C program. When a Fortran function is called as a subroutine, the return value is the first parameter in the calling sequence. You can use this feature to call a BLAS function from C.

The following example shows how a call to a Fortran function as a subroutine converts to a call from C and the hidden parameter result gets exposed:

Normal Fortran function call: `result = cdotc( n, x, 1, y, 1 )`

A call to the function as a subroutine: `call cdotc( result, n, x, 1, y, 1)`

A call to the function from C: `cdotc( &result, &n, x, &one, y, &one )`

NOTE. Intel(R) Math Kernel Library (Intel(R) MKL) has both upper-case and lower-case entry points in the Fortran-style (case-insensitive) BLAS, with or without the trailing underscore. So, all these names are equivalent and acceptable: `cdotc`, `CDOTC`, `cdotc_`, and `CDOTC_`.

The above example shows one of the ways to call several level 1 BLAS functions that return complex values from your C and C++ applications. An easier way is to use the CBLAS interface. For instance, you can call the same function using the CBLAS interface as follows:

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

NOTE. The complex value comes last on the argument list in this case.

The following examples show use of the Fortran-style BLAS interface from C and C++, as well as the CBLAS (C language) interface:

- Example "Calling a Complex BLAS Level 1 Function from C"
- Example "Calling a Complex BLAS Level 1 Function from C++"
- Example "Using CBLAS Interface Instead of Calling BLAS Directly from C"

Example "Calling a Complex BLAS Level 1 Function from C"

The example below illustrates a call from a C program to the complex BLAS Level 1 function `zdotc()`. This function computes the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the structure c.

```
#include "mkl.h"
#define N 5
void main()
{
   MKL_int n = N, inca = 1, incb = 1, i;
   MKL_Complex16 a[N], b[N], c;

   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( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

Calling a Complex BLAS Level 1 Function from C++

Below is the C++ implementation:

```
#include <complex>
#include <iostream>
#define MKL_Complex16 std::complex<double>
#include "mkl.h"

#define N 5

int main()
{
 int n, inca = 1, incb = 1, i;
 std::complex<double> a[N], b[N], c;
 n = N;

 for( i = 0; i < n; i++ ){
  a[i] = std::complex<double>(i,i*2.0);
  b[i] = std::complex<double>(n-i,i*2.0);
 }
 zdotc(&c, &n, a, &inca, b, &incb );
 std::cout << "The complex dot product is: " << c << std::endl;
 return 0;
}
```

Example “Using CBLAS Interface Instead of Calling BLAS Directly from C”

This example uses CBLAS:

```
#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( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
}
```

Support for Boost uBLAS Matrix-matrix Multiplication

If you are used to uBLAS, you can perform BLAS matrix-matrix multiplication in C++ using the Intel(R) Math Kernel Library (Intel(R) MKL) substitution of Boost uBLAS functions. uBLAS is the Boost C++ open-source library that provides BLAS functionality for dense, packed, and sparse matrices. The library uses an expression template technique for passing expressions as function arguments, which enables evaluating vector and matrix expressions in one pass without temporary matrices. uBLAS provides two modes:

- Debug (safe) mode, default.
  Checks types and conformance.
- Release (fast) mode.
  Does not check types and conformance. To enable this mode, use the `NDEBUG` preprocessor symbol.

The documentation for the Boost uBLAS is available at www.boost.org.

Intel MKL provides overloaded `prod()` functions for substituting uBLAS dense matrix-matrix multiplication with the Intel MKL `gemm` calls. Though these functions break uBLAS expression templates and introduce temporary matrices, the performance advantage can be considerable for matrix sizes that are not too small (roughly, over 50).

You do not need to change your source code to use the functions. To call them:

- Include the header file `mkl_boost_ublas_matrix_prod.hpp` in your code (from the Intel MKL include directory)
- Add appropriate Intel MKL libraries to the link line.

The list of expressions that are substituted follows:

```
prod( m1, m2 )

prod( trans(m1), m2 )

prod( trans(conj(m1)), m2 )

prod( conj(trans(m1)), m2 )

prod( m1, trans(m2) )

prod( trans(m1), trans(m2) )

prod( trans(conj(m1)), trans(m2) )

prod( conj(trans(m1)), trans(m2) )

prod( m1, trans(conj(m2)) )

prod( trans(m1), trans(conj(m2)) )
```

```
prod( trans(conj(m1)), trans(conj(m2)) )
prod( conj(trans(m1)), trans(conj(m2)) )
prod( m1, conj(trans(m2)) )
prod( trans(m1), conj(trans(m2)) )
prod( trans(conj(m1)), conj(trans(m2)) )
prod( conj(trans(m1)), conj(trans(m2)) )
```

These expressions are substituted in the *release* mode only (with `NDEBUG` preprocessor symbol defined). Supported uBLAS versions are Boost 1.34.1, 1.35.0, 1.36.0, and 1.37.0. To get them, visit www.boost.org.

A code example provided in the `<mkl directory>/examples/ublas/source/sylvester.cpp` file illustrates usage of the Intel MKL uBLAS header file for solving a special case of the Sylvester equation.

To run the Intel MKL ublas examples, specify the `BOOST_ROOT` parameter in the `make` command, for instance, when using Boost version 1.37.0:

```
make libia32 BOOST_ROOT = <your_path>/boost_1_37_0
```

See Also

- Using Code Examples

Invoking Intel(R) Math Kernel Library Functions from Java* Applications

Intel MKL Java* Examples

To demonstrate binding with Java, Intel(R) Math Kernel Library (Intel(R) MKL) includes a set of Java examples in the following directory:

`<mkl directory>/examples/java`.

The examples are provided for the following MKL functions:

- `?gemm`, `?gemv`, and `?dot` families from CBLAS
- The complete set of non-cluster FFT functions
- ESSL[1] -like functions for one-dimensional convolution and correlation
- VSL Random Number Generators (RNG), except user-defined ones and file subroutines
- VML functions, except `GetErrorCallBack`, `SetErrorCallBack`, and `ClearErrorCallBack`

You can see the example sources in the following directory:

`<mkl directory>/examples/java/examples`.

The examples are written in Java. They demonstrate usage of the MKL functions with the following variety of data:

- 1- and 2-dimensional data sequences
- Real and complex types of the data
- Single and double precision

However, the wrappers, used in the examples, do not:

- Demonstrate the use of large arrays (>2 billion elements)
- Demonstrate processing of arrays in native memory

- Check correctness of function parameters
- Demonstrate performance optimizations

The examples use the Java Native Interface (JNI* developer framework) to bind with Intel MKL. The JNI documentation is available from http://java.sun.com/javase/6/docs/technotes/guides/jni/.

The Java example set includes JNI wrappers that perform the binding. The wrappers do not depend on the examples and may be used in your Java applications. The wrappers for CBLAS, FFT, VML, VSL RNG, and ESSL-like convolution and correlation functions do not depend on each other.

To build the wrappers, just run the examples. The makefile builds the wrapper binaries. After running the makefile, you can run the examples, which will determine whether the wrappers were built correctly. As a result of running the examples, the following directories will be created in `<`*`mkl directory`*`>/examples/java`:

- `docs`
- `include`
- `classes`
- `bin`
- `_results`

The directories `docs` `include` `classes` and `bin` will contain the wrapper binaries and documentation; the directory `_results` will contain the testing results.

For a Java programmer, the wrappers are the following Java classes:

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

Documentation for the particular wrapper and example classes will be generated from the Java sources while building and running the examples. To browse the documentation, open the index file in the `docs` directory (created by the build script):

`<`*`mkl directory`*`>/examples/java/docs/index.html`.

The Java wrappers for CBLAS, VML, VSL RNG, and FFT establish the interface that directly corresponds to the underlying native functions, so you can refer to the Intel MKL Reference Manual for their functionality and parameters. Interfaces for the ESSL-like functions are described in the generated documentation for the `com.intel.mkl.ESSL` class.

Each wrapper consists of the interface part for Java and JNI stub written in C. You can find the sources in the following directory:

`<`*`mkl directory`*`>/examples/java/wrappers`.

Both Java and C parts of the wrapper for CBLAS and VML demonstrate the straightforward approach, which you may use to cover additional CBLAS functions.

The wrapper for FFT is more complicated because it needs to support the lifecycle for FFT descriptor objects. To compute a single Fourier transform, an application needs to call the FFT software several times with the same copy of the native FFT descriptor. The wrapper provides the handler class to hold the native descriptor, while the virtual machine runs Java bytecode.

The wrapper for VSL RNG is similar to the one for FFT. The wrapper provides the handler class to hold the native descriptor of the stream state.

The wrapper for the convolution and correlation functions mitigates the same difficulty of the VSL interface, which assumes a similar lifecycle for "task descriptors". The wrapper utilizes the ESSL-like interface for those functions, which is simpler for the case of 1-dimensional data. The JNI stub additionally encapsulates the MKL functions into the ESSL-like wrappers written in C and so "packs" the lifecycle of a task descriptor into a single call to the native method.

The wrappers meet the JNI Specification versions 1.1 and 5.0 and should work with virtually every modern implementation of Java.

The examples and the Java part of the wrappers are written for the Java language described in "The Java Language Specification (First Edition)" and extended with the feature of "inner classes" (this refers to late 1990s). This level of language version is supported by all versions of the Sun Java Development Kit* (JDK*) developer toolkit and compatible implementations starting from version 1.1.5, or by all modern versions of Java.

The level of C language is "Standard C" (that is, C89) with additional assumptions about integer and floating-point data types required by the Intel MKL interfaces and the JNI header files. That is, the native `float` and `double` data types must be the same as JNI `jfloat` and `jdouble` data types, respectively, and the native `int` must be 4 bytes long.

[1.] IBM Engineering Scientific Subroutine Library (ESSL*).

See Also

- Running the Examples

Running the Java* Examples

The Java examples support all the C and C++ compilers that the Intel(R) Math Kernel Library (Intel(R) MKL) does. The makefile intended to run the examples also needs the make utility, which is typically provided with the Linux* OS distribution.

To run Java examples, the JDK* developer toolkit is required for compiling and running Java code. A Java implementation must be installed on the computer or available via the network. You may download the JDK from the vendor website.

The examples should work for all versions of JDK. However, they were tested only with the following Java implementation s for all the supported architectures:

- J2SE* SDK 1.4.2, JDK 5.0 and 6.0 from Sun Microsystems, Inc. (http://sun.com/).
- JRockit* JDK 1.4.2 and 5.0 from Oracle Corporation (http://oracle.com/).

Note that the Java run-time environment* (JRE*) system, which may be pre-installed on your computer, is not enough. You need the JDK* developer toolkit that supports the following set of tools:

- java
- javac
- javah
- javadoc

To make these tools available for the examples makefile, set the `JAVA_HOME` environment variable and add the JDK binaries directory to the system `PATH`, for example , using the bash shell:

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

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

You may also need to clear the `JDK_HOME` environment variable, if it is assigned a value:

```
unset JDK_HOME
```

To start the examples, use the makefile found in the Intel MKL Java examples directory:

```
make {soia32|sointel64 |libia32|libintel64 } [function=...] [compiler=...]
```

If you type the make command and omit the target (for example, `soia32`), the makefile prints the help info, which explains the targets and parameters.

For the examples list, see the `examples.lst` file in the Java examples directory.

Known Limitations of the Java* Examples

This section explains limitations of Java examples.

Functionality

Some Intel(R) Math Kernel Library (Intel(R) MKL) functions may fail to work if called from the Java environment by using a wrapper, like those provided with the Intel MKL Java examples. Only those specific CBLAS, FFT, VML, VSL RNG, and the convolution/correlation functions listed in the Intel MKL Java Examples section were tested with the Java environment. So, you may use the Java wrappers for these CBLAS, FFT, VML, VSL RNG, and convolution/correlation functions in your Java applications.

Performance

The Intel MKL functions must work faster than similar functions written in pure Java. However, the main goal of these wrappers is to provide code examples, not maximum performance. So, an Intel MKL function called from a Java application will probably work slower than the same function called from a program written in C/C++ or Fortran.

Known bugs

There are a number of known bugs in Intel MKL (identified in the Release Notes), as well as incompatibilities between different versions of JDK. The examples and wrappers include workarounds for these problems. Look at the source code in the examples and wrappers for comments that describe the workarounds.

Coding Tips

7

This section discusses programming with the Intel(R) Math Kernel Library (Intel(R) MKL) to provide coding tips that meet certain, specific needs, such as numerical stability.

Aligning Data for Consistent Results

Routines in the Intel(R) Math Kernel Library (Intel(R) MKL) may return different results from run-to-run on the same system. This is usually due to a change in the order in which floating-point operations are performed. The two most influential factors are array alignment and parallelism. Array alignment can determine how internal loops order floating-point operations. Non-deterministic parallelism may change the order in which computational tasks are executed. While these results may differ, they should still fall within acceptable computational error bounds. To better assure identical results from run-to-run, do the following:

- Align input arrays on 16-byte boundaries
- Run Intel MKL in the sequential mode

To align input arrays on 16-byte boundaries, use `mkl_malloc()` in place of system provided memory allocators, as shown in the code example below. Sequential mode of Intel MKL removes the influence of non-deterministic parallelism.

Aligning Addresses on 16-byte Boundaries

```
// ******* C language *******
...
#include <stdlib.h>
...
void *darray;
int workspace;
...
// Allocate workspace aligned on 16-bit boundary
darray = mkl_malloc( sizeof(double)*workspace, 16 );
...
// call the program using MKL
mkl_app( darray );
...
// Free workspace
mkl_free( darray );


! ******* Fortran language *******
...
double precision darray
pointer (p_wrk,darray(1))
```

```
integer workspace
...
! Allocate workspace aligned on 16-bit boundary
p_wrk = mkl_malloc( 8*workspace, 16 )
...
! call the program using MKL
call mkl_app( darray )
...
! Free workspace
call mkl_free(p_wrk)
```

Working with the Intel(R) Math Kernel Library Cluster Software

8

Optimization Notice

The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

Linking with ScaLAPACK and Cluster FFTs

The Intel(R) Math Kernel Library (Intel(R) MKL) ScaLAPACK and Cluster FFTs support MPI implementations identified in the *Intel MKL Release Notes*.

To link a program that calls ScaLAPACK or Cluster FFTs, you need to know how to link a message-passing interface (MPI) application first.

Use mpi scripts to do this. For example, mpicc or mpif77 are C or FORTRAN 77 scripts, respectively, that use the correct MPI header files. The location of these scripts and the MPI library depends on your MPI implementation. For example, for the default installation of MPICH, `/opt/mpich/bin/mpicc` and `/opt/mpich/bin/mpif77` are the compiler scripts and `/opt/mpich/lib/libmpich.a` is the MPI library.

Check the documentation that comes with your MPI implementation for implementation-specific details of linking.

To link with the Intel(R) Math Kernel Library (Intel(R) MKL) ScaLAPACK and/or Cluster FFTs, use the following general form:

```
<<MPI> linker script>  <files to link>                  \
-L <MKL path> [-Wl,--start-group] <MKL cluster library>        \
<BLACS> <MKL core libraries> [-Wl,--end-group]
```

where the placeholders stand for paths and libraries as explained in the following table:

<MKL cluster library>	One of ScaLAPACK or Cluster FFT libraries for the appropriate architecture, which are listed in Directory Structure in Detail. For example, for IA-32 architecture, it is either `-lmkl_scalapack_core` or `-lmkl_cdft_core`.
<BLACS>	The BLACS library corresponding to your architecture, programming interface (LP64 or ILP64), and MPI version. Available BLACS libraries are listed in Directory Structure in Detail. For example, for the IA-32 architecture, choose one of `-lmkl_blacs`, `-lmkl_blacs_intelmpi`, or `-lmkl_blacs_openmpi`, depending on the MPI version you use; specifically, for Intel MPI 3.x, choose `-lmkl_blacs_intelmpi`.
<MKL core libraries>	*<MKL LAPACK & MKL kernel libraries>* for ScaLAPACK, and *<MKL kernel libraries>* for Cluster FFTs.
<MKL kernel libraries>	Processor optimized kernels, threading library, and system library for threading support, linked as described in Listing Libraries on a Link Line.
<MKL LAPACK & kernel libraries>	The LAPACK library and *<MKL kernel libraries>*.
<MPI>	One of several MPI implementations (MPICH, Intel MPI, and so on).
<<MPI> linker script>	A linker script that corresponds to the MPI version. For instance, for Intel MPI 3.x, use *<Intel MPI 3.x linker script>*.

For example, if you are using Intel MPI 3.x, want to statically use the LP64 interface with ScaLAPACK, and have only one MPI process per core (and thus do not use threading), specify the following linker options:

```
-L$MKLPATH -I$MKLINCLUDE -Wl,--start-group $MKLPATH/libmkl_scalapack_lp64.a
$MKLPATH/libmkl_blacs_intelmpi_lp64.a $MKLPATH/libmkl_intel_lp64.a
$MKLPATH/libmkl_sequential.a $MKLPATH/libmkl_core.a -static_mpi -Wl,--end-group -lpthread
-lm
```

NOTE. Grouping symbols `-Wl,--start-group` and `-Wl,--end-group` are required for static linking.

TIP. Use the Web-based Linking Advisor to quickly choose the appropriate set of *<MKL cluster Library>*, *<BLACS>*, and *<MKL core libraries>*.

See Also

- Linking Your Application with the Intel(R) Math Kernel Library
- Examples for Linking with ScaLAPACK and Cluster FFT

Setting the Number of Threads

The OpenMP* software responds to the environment variable `OMP_NUM_THREADS`. Intel(R) Math Kernel Library (Intel(R) MKL) also has other mechanisms to set the number of threads, such as the `MKL_NUM_THREADS` or `MKL_DOMAIN_NUM_THREADS` environment variables (see Using Additional Threading Control).

Make sure that the relevant environment variables have the same and correct values on all the nodes. Intel MKL versions 10.0 and higher no longer set the default number of threads to one, but depend on the OpenMP libraries used with the compiler to set the default number. For the threading layer based on the Intel(R) compiler (`libmkl_intel_thread.a`), this value is the number of CPUs according to the OS.

CAUTION. Avoid over-prescribing the number of threads, which may occur, for instance, when the number of MPI ranks per node and the number of threads per node are both greater than one. The product of MPI ranks per node and the number of threads per node should not exceed the number of physical cores per node.

The best way to set an environment variable, such as `OMP_NUM_THREADS`, is your login environment. Remember that changing this value on the head node and then doing your run, as you do on a shared-memory (SMP) system, does not change the variable on all the nodes because `mpirun` starts a fresh default shell on all of the nodes. To change the number of threads on all the nodes, in `.bashrc`, add a line at the top, as follows:

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

You can run multiple CPUs per node using MPICH. To do this, build MPICH to enable multiple CPUs per node. Be aware that certain MPICH applications may fail to work perfectly in a threaded environment (see the Known Limitations section in the *Release Notes*. If you encounter problems with MPICH and setting of the number of threads is greater than one, first try setting the number of threads to one and see whether the problem persists.

See Also

Using Shared Libraries

All needed shared libraries must be visible on all the nodes at run time. To achieve this, point these libraries by the `LD_LIBRARY_PATH` environment variable in the `.bashrc` file.

If the Intel(R) Math Kernel Library (Intel(R) MKL) is installed only on one node, link statically when building your Intel MKL applications rather than use shared libraries.

The Intel(R) compilers or GNU compilers can be used to compile a program that uses Intel MKL. However, make sure that the MPI implementation and compiler match up correctly.

Building ScaLAPACK Tests

To build ScaLAPACK tests,

- For the IA-32 architecture, add `libmkl_scalapack_core.a` to your link command.
- For the Intel(R) 64 architecture, add `libmkl_scalapack_lp64.a` or `libmkl_scalapack_ilp64.a`, depending upon the desired interface.

Examples for Linking with ScaLAPACK and Cluster FFT

This section provides examples of linking with ScaLAPACK and Cluster FFT.

Note that a binary linked with ScaLAPACK runs the same way as any other MPI application (refer to the documentation that comes with your MPI implementation). For instance, the script `mpirun` is used in the case of MPICH2 and OpenMPI, and a number of MPI processes is set by `-np`. In the case of MPICH 2.0 and all Intel MPIs, start the daemon before running your application; the execution is driven by the script `mpiexec`.

For further linking examples, see the support website for Intel products at http://www.intel.com/software/products/support/.

See Also

- Directory Structure in Detail

Examples for Linking a C Application

These examples illustrate linking of an application whose main module is in C under the following conditions:

- MPICH2 1.0.7 or higher is installed in `/opt/mpich`.
- $MKLPATH is a user-defined variable containing `<`*`mkl_directory`*`>/lib/ia32`.
- You use the Intel(R) C++ Compiler 10.0 or higher.

To link with ScaLAPACK for a cluster of systems based on the IA-32 architecture, use the following link line:

```
/opt/mpich/bin/mpicc  <user files to link>       \
   -L$MKLPATH                                    \
   -lmkl_scalapack_core                          \
   -lmkl_blacs_intelmpi                          \
   -lmkl_intel -lmkl_intel_thread -lmkl_core     \
   -liomp5 -lpthread
```

To link with Cluster FFT for a cluster of systems based on the IA-32 architecture, use the following link line:

```
/opt/mpich/bin/mpicc  <user files to link> \
   -Wl,--start-group                       \
   $MKLPATH/libmkl_cdft_core.a             \
   $MKLPATH/libmkl_blacs_intelmpi.a        \
   $MKLPATH/libmkl_intel.a                 \
   $MKLPATH/libmkl_intel_thread.a          \
   $MKLPATH/libmkl_core.a                  \
   -Wl,--end-group                         \
   -liomp5 -lpthread
```

See Also

- Linking with ScaLAPACK and Cluster FFTs

Examples for Linking a Fortran Application

These examples illustrate linking of an application whose main module is in Fortran under the following conditions:

- Intel MPI 3.0 is installed in `/opt/intel/mpi/3.0`.
- $MKLPATH is a user-defined variable containing `<`*`mkl directory`*`>/lib/64`.
- You use the Intel(R) Fortran Compiler 10.0 or higher.

To link with ScaLAPACK for a cluster of systems based on the Intel(R) 64 architecture, use the following link line:

```
/opt/intel/mpi/3.0/bin/mpiifort <user files to link> \
   -L$MKLPATH                                        \
   -lmkl_scalapack_lp64                              \
   -lmkl_blacs_intelmpi_lp64                         \
   -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core    \
   -liomp5 -lpthread
```

To link with Cluster FFT for a cluster of systems based on the Intel(R) 64 architecture, use the following link line:

```
/opt/intel/mpi/3.0/bin/mpiifort <user files to link> \
   -Wl,--start-group                                 \
   $MKLPATH/libmkl_cdft_core.a                       \
   $MKLPATH/libmkl_blacs_intelmpi_ilp64.a            \
   $MKLPATH/libmkl_intel_ilp64.a                     \
   $MKLPATH/libmkl_intel_thread.a                    \
   $MKLPATH/libmkl_core.a                            \
   -Wl,--end-group                                   \
   -liomp5 -lpthread
```

See Also

- Linking with ScaLAPACK and Cluster FFTs

Programming with Intel(R) Math Kernel Library in the Eclipse* Integrated Development Environment (IDE)

9

Configuring the Eclipse* IDE CDT to Link with Intel(R) Math Kernel Library

This section explains how to configure the Eclipse* Integrated Development Environment (IDE) C/C++ Development Tools (CDT) 3.x and 4.0 to link with Intel(R) Math Kernel Library (Intel(R) MKL).

TIP. After linking your CDT with Intel MKL, you can benefit from the Eclipse-provided *code assist* feature. See Code/Context Assist description in the Eclipse IDE Help.

Configuring the Eclipse* IDE CDT 3.x

To configure Eclipse* IDE C/C++ Development Tools (CDT) 3.x to link with Intel(R) Math Kernel Library (Intel(R) MKL), follow the instructions below:

- For Standard Make projects:
  - Go to **C/C++ Include Paths and Symbols** property page and set the Intel MKL include path to <*mkl directory*>/include.
  - Go to **C/C++ Project Paths** > **Libraries** and set the Intel MKL libraries to link with your applications, for example, <mkl directory>/lib/intel64/libmkl_intel_lp64.a, <mkl directory>/lib/intel64/libmkl_intel_thread.a, and <*mkl directory*>/lib/intel64/libmkl_core.a.

  Note that with the Standard Make, the above settings are needed for the CDT internal functionality only. The compiler/linker will not automatically pick up these settings and you will still have to specify them directly in the makefile.
- For Managed Make projects, you can specify settings for a particular build. To do this:
  - Go to **C/C++ Build** > **Tool Settings**. All the settings you need to specify are on this page. Names of the particular settings depend on the compiler integration and therefore are not explained below.

- If the compiler integration supports include path options, set the Intel MKL include path to <*mkl directory*>/include.
- If the compiler integration supports library path options, set a path to the Intel MKL libraries for the target architecture, such as <*mkl directory*>/lib/intel64.
- Specify the names of the Intel MKL libraries to link with your application, for example, mkl_intel_lp64, mkl_intel_thread_lp64, mkl_core, and iomp5 (compilers typically require library names rather than library file names, so omit the "lib" prefix and "a" extension).

See Also

- Selecting Libraries to Link

Configuring the Eclipse* IDE CDT 4.0

Before configuring Eclipse IDE C/C++ Development Tools (CDT) 4.0, make sure to turn on the automatic makefile generation.

To configure Eclipse CDT 4.0 to link with Intel MKL, follow the instructions below:

1. If the tool-chain/compiler integration supports include path options, go to **C/C++ General** > **Paths and Symbols** > **Includes** and set the Intel(R) Math Kernel Library (Intel(R) MKL) include path, that is, <*mkl directory*>/include.
2. If the tool-chain/compiler integration supports library path options, go to **C/C++ General** > **Paths and Symbols** > **Library Paths** and set the Intel MKL library path for the target architecture, such as <*mkl directory*>/lib/intel64.
3. Go to **C/C++ Build** > **Settings** > **Tool Settings** and specify the names of the Intel MKL libraries to link with your application, for example, mkl_intel_lp64, mkl_intel_thread_lp64, mkl_core, and iomp5 (compilers typically require library names rather than library file names, so omit the "lib" prefix and "a" extension). To learn how to choose the libraries, see Selecting Libraries to Link. The name of the particular setting where libraries are specified depends upon the compiler integration.

Getting Assistance for Programming in the Eclipse* IDE

Intel MKL provides an Eclipse* IDE plug-in (com.intel.mkl.help) that contains the Intel(R) Math Kernel Library (Intel(R) MKL) Reference Manual (see High-level Directory Structure for the plug-in location after the library installation). To install the plug-in, do one of the following:

- Use the Eclipse IDE Update Manager (recommended).

  To invoke the Manager, use **Help > Software Updates** command in your Eclipse IDE.
- Copy the plug-in to the plugins folder of your Eclipse IDE directory.

  In this case, if you use earlier C/C++ Development Tools (CDT) versions (3.x, 4.x), delete or rename the index subfolder in the eclipse/configuration/org.eclipse.help.base folder of your Eclipse IDE to avoid delays in Index updating.

The following Intel MKL features assist you while programming in the Eclipse* IDE:

- The Intel MKL Reference Manual viewable from within the IDE
- Eclipse Help search tuned to target the Intel Web sites
- Code/Content Assist in the Eclipse IDE CDT

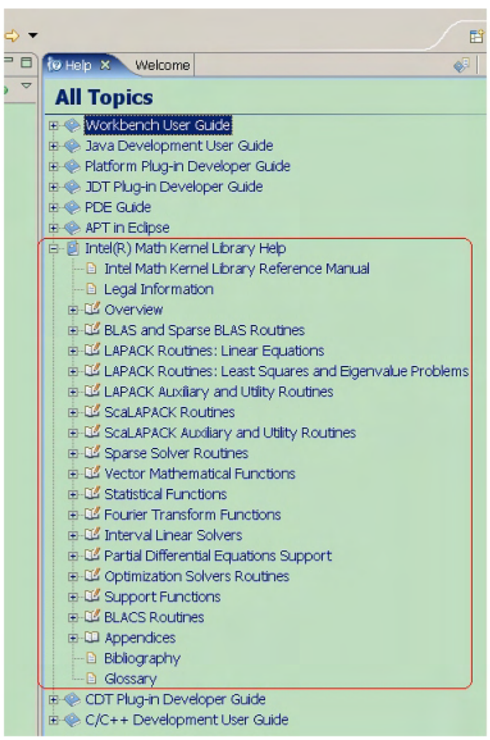
The Intel MKL plug-in for Eclipse IDE provides the first two features.

The last feature is native to the Eclipse IDE CDT. See the Code Assist description in Eclipse IDE Help for details.

Viewing the Intel(R) Math Kernel Library Reference Manual in the Eclipse* IDE

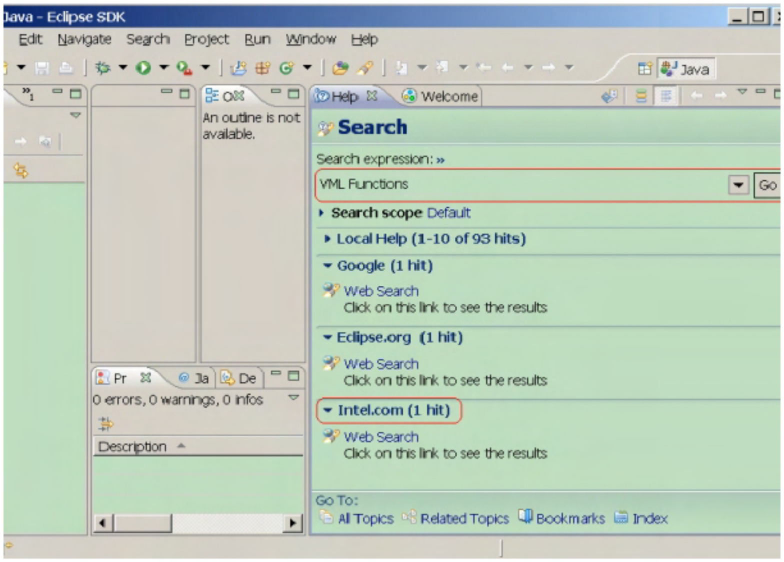
To view the Reference Manual, in Eclipse,

1. Select **Help** > **Help Contents** from the menu.
2. In the Help tab, under **All Topics**, click **Intel(R) Math Kernel Library Help**.
3. In the Help tree that expands, click **Intel Math Kernel Library Reference Manual**.
4. The Intel MKL Help Index is also available in Eclipse, and the Reference Manual is included in the Eclipse Help search.



Searching the Intel Web Site from the Eclipse* IDE

The Intel(R) Math Kernel Library (Intel(R) MKL) plug-in tunes Eclipse Help search to target http://www.intel.com so that when you are connected to the Internet and run a search from the Eclipse Help pane, the search hits at the site are shown through a separate link. The following figure shows search results for "VML Functions" in Eclipse Help. In the figure, 1 hit means an entry hit to the respective site.
Click "Intel.com (1 hit)" to open the list of actual hits to the Intel Web site.


Java - Eclipse SDK
Edit Navigate Search Project Run Window Help
Java
An outline is not available.
Help
Welcome
Search
Search expression:
VML Functions
Go
Search scope Default
Local Help (1-10 of 93 hits)
Google (1 hit)
Web Search
Click on this link to see the results
Eclipse.org (1 hit)
Web Search
Click on this link to see the results
Intel.com (1 hit)
Web Search
Click on this link to see the results
0 errors, 0 warnings, 0 infos
Description
Go To:
All Topics
Related Topics
Bookmarks
Index

LINPACK and MP LINPACK Benchmarks

10

Optimization Notice

The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors.

While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

Intel recommends that you evaluate other library products to determine which best meets your requirements.

Intel(R) Optimized LINPACK Benchmark for Linux* OS

Intel(R) Optimized LINPACK Benchmark is a generalization of the LINPACK 1000 benchmark. It solves a dense (`real*8`) system of linear equations ($Ax=b$), measures the amount of time it takes to factor and solve the system, converts that time into a performance rate, and tests the results for accuracy. The generalization is in the number of equations (N) it can solve, which is not limited to 1000. It uses partial pivoting to assure the accuracy of the results.

Do not use this benchmark to report LINPACK 100 performance because that is a compiled-code only benchmark. This is a shared-memory (SMP) implementation which runs on a single platform. Do not confuse this benchmark with :

- MP LINPACK, which is a distributed memory version of the same benchmark.
- LINPACK, the library, which has been expanded upon by the LAPACK library.

Intel provides optimized versions of the LINPACK benchmarks to help you obtain high LINPACK benchmark results on your genuine Intel(R) processor systems more easily than with the High Performance Linpack (HPL) benchmark. Use this package to benchmark your SMP machine.

Additional information on this software as well as other Intel(R) software performance products is available at http://www.intel.com/software/products/.

Contents of the Intel(R) Optimized LINPACK Benchmark

The Intel Optimized LINPACK Benchmark for Linux* OS contains the following files, located in the ./benchmarks/linpack/ subdirectory in the Intel(R) Math Kernel Library (Intel(R) MKL) directory:

File in ./benchmarks/linpack/	**Description**
linpack_xeon32	The 32-bit program executable for a system based on Intel(R) Xeon(R) processor or Intel(R) Xeon(R) processor MP with or without Streaming SIMD Extensions 3 (SSE3).
linpack_xeon64	The 64-bit program executable for a system with Intel(R) Xeon(R) processor using Intel(R) 64 architecture.
runme_xeon32	A sample shell script for executing a pre-determined problem set for linpack_xeon32. OMP_NUM_THREADS set to 2 processors.
runme_xeon64	A sample shell script for executing a pre-determined problem set for linpack_xeon64. OMP_NUM_THREADS set to 4 processors.
lininput_xeon32	Input file for pre-determined problem for the runme_xeon32 script.
lininput_xeon64	Input file for pre-determined problem for the runme_xeon64 script.
lin_xeon32.txt	Result of the runme_xeon32 script execution.
lin_xeon64.txt	Result of the runme_xeon64 script execution.
help.lpk	Simple help file.
xhelp.lpk	Extended help file.

See Also

- High-level Directory Structure

Running the Software

To obtain results for the pre-determined sample problem sizes on a given system, type one of the following, as appropriate:

```
./runme_xeon32
```

```
./runme_xeon64
```

To run the software for other problem sizes, see the extended help included with the program. Extended help can be viewed by running the program executable with the -e option:

```
./xlinpack_xeon32 -e
```

```
./xlinpack_xeon64 -e
```

The pre-defined data input files `lininput_xeon32` and `lininput_xeon64` are provided merely as examples. Different systems have different number of processors or amount of memory and thus require new input files. The extended help can be used for insight into proper ways to change the sample input files.

Each input file requires at least the following amount of memory:

`lininput_xeon32`	2 GB
`lininput_xeon64`	16 GB

If the system has less memory than the above sample data input requires, you may need to edit or create your own data input files, as explained in the extended help.

Each sample script, in particular, uses the `OMP_NUM_THREADS` environment variable to set the number of processors it is targeting. To optimize performance on a different number of physical processors, change that line appropriately. If you run the Intel Optimized LINPACK Benchmark without setting the number of threads, it will default to the number of cores according to the OS. You can find the settings for this environment variable in the `runme_*` sample scripts. If the settings do not already match the situation for your machine, edit the script.

Known Limitations of the Intel(R) Optimized LINPACK Benchmark

The following limitations are known for the Intel Optimized LINPACK Benchmark for Linux* OS:

- Intel Optimized LINPACK Benchmark is threaded to effectively use multiple processors. So, in multi-processor systems, best performance will be obtained with Hyper-Threading technology turned off, which ensures that the operating system assigns threads to physical processors only.
- If an incomplete data input file is given, the binaries may either hang or fault. See the sample data input files and/or the extended help for insight into creating a correct data input file.

Intel(R) Optimized MP LINPACK Benchmark for Clusters

Overview of the Intel(R) Optimized MP LINPACK Benchmark for Clusters

The Intel(R) Optimized MP LINPACK Benchmark for Clusters is based on modifications and additions to HPL 2.0 from Innovative Computing Laboratories (ICL) at the University of Tennessee, Knoxville (UTK). The Intel Optimized MP LINPACK Benchmark for Clusters can be used for Top 500 runs (see http://www.top500.org). To use the benchmark you need be intimately familiar with the HPL distribution and usage. The Intel Optimized MP LINPACK Benchmark for Clusters provides some additional enhancements and bug fixes designed to make the HPL usage more convenient, as well as explain Intel(R) Message-Passing Interface (MPI) settings that may enhance performance. The `./benchmarks/mp_linpack` directory adds techniques to minimize search times frequently associated with long runs.

The Intel(R) Optimized MP LINPACK Benchmark for Clusters is an implementation of the Massively Parallel MP LINPACK benchmark by means of HPL code. It solves a random dense (`real*8`) system of linear equations (*Ax*=*b*), measures the amount of time it takes to factor and solve the system, converts that time into a performance rate, and tests the results for accuracy. You can solve any size (*N*) system of equations that fit into memory. The benchmark uses full row pivoting to ensure the accuracy of the results.

Use the Intel Optimized MP LINPACK Benchmark for Clusters on a distributed memory machine. On a shared memory machine, use the Intel Optimized LINPACK Benchmark.

Intel provides optimized versions of the LINPACK benchmarks to help you obtain high LINPACK benchmark results on your systems based on genuine Intel(R) processors more easily than with the HPL benchmark. Use the Intel(R) Optimized MP LINPACK Benchmark to benchmark your cluster. The prebuilt binaries require that you first install Intel(R) MPI 3.x be installed on the cluster. The run-time version of Intel MPI is free and can be downloaded from www.intel.com/software/products/.

The Intel package includes software developed at the University of Tennessee, Knoxville, Innovative Computing Laboratories and neither the University nor ICL endorse or promote this product. Although HPL 2.0 is redistributable under certain conditions, this particular package is subject to the Intel(R) Math Kernel Library (Intel(R) MKL) license.

Intel MKL has introduced a new functionality into MP LINPACK, which is called a *hybrid* build, while continuing to support the older version. The term " *hybrid* " refers to special optimizations added to take advantage of mixed OpenMP*/MPI parallelism.

If you want to use one MPI process per node and to achieve further parallelism by means of OpenMP, use the hybrid build. In general, the hybrid build is useful when the number of MPI processes per core is less than one. If you want to rely exclusively on MPI for parallelism and use one MPI per core, use the non-hybrid build.

In addition to supplying certain hybrid prebuilt binaries, Intel MKL supplies some hybrid prebuilt libraries for Intel(R) MPI to take advantage of the additional OpenMP* optimizations.

If you wish to use an MPI version other than Intel MPI, you can do so by using the MP LINPACK source provided. You can use the source to build a non-hybrid version that may be used in a hybrid mode, but it would be missing some of the optimizations added to the hybrid version.

Non-hybrid builds are the default of the source code makefiles provided. In some cases, the use of the hybrid mode is required for external reasons. If there is a choice, the non-hybrid code may be faster. To use the non-hybrid code in a hybrid mode, use the threaded version of Intel MKL BLAS, link with a thread-safe MPI, and call function `MPI_init_thread()` so as to indicate a need for MPI to be thread-safe.

Intel MKL also provides prebuilt binaries that are dynamically linked against Intel MPI libraries.

NOTE. Performance of statically and dynamically linked prebuilt binaries may be different. The performance of both depends on the version of Intel MPI you are using.
You can build binaries statically linked against a particular version of Intel MPI by yourself.

Contents of the Intel(R) Optimized MP LINPACK Benchmark for Clusters

The Intel Optimized MP LINPACK Benchmark for Clusters (MP LINPACK Benchmark) includes the HPL 2.0 distribution in its entirety, as well as the modifications delivered in the files listed in the table below and located in the `./benchmarks/mp_linpack/` subdirectory in the Intel(R) Math Kernel Library (Intel(R) MKL) directory.

Directory/File in `./benchmarks/mp_linpack/`	**Contents**
`testing/ptest/HPL_pdtest.c`	HPL 2.0 code modified to display captured `DGEMM` information in `ASYOUGO2_DISPLAY` if it was captured (for details, see New Features).
`src/blas/HPL_dgemm.c`	HPL 2.0 code modified to capture `DGEMM` information, if desired, from `ASYOUGO2_DISPLAY`.

Directory/File in ./benchmarks/mp_linpack/	Contents
src/grid/HPL_grid_init.c	HPL 2.0 code modified to do additional grid experiments originally not in HPL 2.0.
src/pgesv/HPL_pdgesvK2.c	HPL 2.0 code modified to do ASYOUGO and ENDEARLY modifications.
src/pgesv/HPL_pdgesv0.c	HPL 2.0 code modified to do ASYOUGO, ASYOUGO2, and ENDEARLY modifications.
testing/ptest/HPL.dat	HPL 2.0 sample HPL.dat modified.
Make.ia32	(New) Sample architecture makefile for processors using the IA-32 architecture and Linux OS.
Make.intel64	(New) Sample architecture makefile for processors using the Intel(R) 64 architecture and Linux OS.
HPL.dat	A repeat of testing/ptest/HPL.dat in the top-level directory.
Next six files are prebuilt executables, readily available for simple performance testing.	
bin_intel/ia32/xhpl_ia32	(New) Prebuilt binary for the IA-32 architecture and Linux OS. Statically linked against Intel(R) MPI 3.2.
bin_intel/ia32/xhpl_ia32_dynamic	(New) Prebuilt binary for the IA-32 architecture and Linux OS. Dynamically linked against Intel(R) MPI 3.2.
bin_intel/intel64/xhpl_intel64	(New) Prebuilt binary for the Intel(R) 64 architecture and Linux OS. Statically linked against Intel(R) MPI 3.2.
bin_intel/intel64/xhpl_intel64_dynamic	(New) Prebuilt binary for the Intel(R) 64 architecture and Linux OS. Dynamically linked against Intel(R) MPI 3.2.
Next six files are prebuilt hybrid executables.	
bin_intel/ia32/xhpl_hybrid_ia32	(New) Prebuilt hybrid binary for the IA-32 architecture and Linux OS. Statically linked against Intel(R) MPI 3.2.
bin_intel/ia32/xhpl_hybrid_ia32_dynamic	(New) Prebuilt hybrid binary for the IA-32 architecture and Linux OS. Dynamically linked against Intel(R) MPI 3.2.
bin_intel/intel64/xhpl_hybrid_intel64	(New) Prebuilt hybrid binary for the Intel(R) 64 architecture and Linux OS. Statically linked against Intel(R) MPI 3.2.
bin_intel/intel64/xhpl_hybrid_intel64_dynamic	(New) Prebuilt hybrid binary for the Intel(R) 64 and Linux OS. Dynamically linked against Intel(R) MPI 3.2.
Next 3 files are prebuilt libraries	
lib_hybrid/ia32/libhpl_hybrid.a	(New) Prebuilt library with the hybrid version of MP LINPACK for the IA-32 architecture and Intel MPI 3.2.

Directory/File in ./benchmarks/mp_linpack/	Contents
lib_hybrid/intel64/libhpl_hybrid.a	(New) Prebuilt library with the hybrid version of MP LINPACK for the Intel(R) 64 architecture and Intel MPI 3.2.
Next 18 files refer to run scripts	
bin_intel/ia32/runme_ia32	(New) Sample run script for the IA-32 architecture and a pure MPI binary statically linked against Intel MPI 3.2.
bin_intel/ia32/runme_ia32_dynamic	(New) Sample run script for the IA-32 architecture and a pure MPI binary dynamically linked against Intel MPI 3.2.
bin_intel/ia32/HPL_serial.dat	(New) Example of an MP LINPACK benchmark input file for a pure MPI binary and the IA-32 architecture.
bin_intel/ia32/runme_hybrid_ia32	(New) Sample run script for the IA-32 architecture and a hybrid binary statically linked against Intel MPI 3.2.
bin_intel/ia32/runme_hybrid_ia32_dynamic	(New) Sample run script for the IA-32 architecture and a hybrid binary dynamically linked against Intel MPI 3.2.
bin_intel/ia32/HPL_hybrid.dat	(New) Example of an MP LINPACK benchmark input file for a hybrid binary and the IA-32 architecture.
bin_intel/intel64/runme_intel64	(New) Sample run script for the Intel(R) 64 architecture and a pure MPI binary statically linked against Intel MPI 3.2.
bin_intel/intel64/runme_intel64_dynamic	(New) Sample run script for the Intel(R) 64 architecture and a pure MPI binary dynamically linked against Intel MPI 3.2.
bin_intel/intel64/HPL_serial.dat	(New) Example of an MP LINPACK benchmark input file for a pure MPI binary and the Intel(R) 64 architecture.
bin_intel/intel64/runme_hybrid_intel64	(New) Sample run script for the Intel(R) 64 architecture and a hybrid binary statically linked against Intel MPI 3.2.
bin_intel/intel64/runme_hybrid_intel64_dynamic	(New) Sample run script for the Intel(R) 64 architecture and a hybrid binary dynamically linked against Intel MPI 3.2.
bin_intel/intel64/HPL_hybrid.dat	(New) Example of an MP LINPACK benchmark input file for a hybrid binary and the Intel(R) 64 architecture.
nodeperf.c	(New) Sample utility that tests the DGEMM speed across the cluster.

See Also

- High-level Directory Structure

Building the MP LINPACK

The MP LINPACK Benchmark contains a few sample architecture makefiles. You can edit them to fit your specific configuration. Specifically:

- Set `TOPdir` to the directory that MP LINPACK is being built in.
- You may set MPI variables, that is, `MPdir`, `MPinc`, and `MPlib`.
- Specify the location of the Intel(R) Math Kernel Library (Intel(R) MKL) and of files to be used (`LAdir`, `LAinc`, `LAlib`).
- Adjust compiler and compiler/linker options.
- Specify the version of MP LINPACK you are going to build (hybrid or non-hybrid) by setting the version parameter for the `make` command. For example:

```
make arch=intel64 version=hybrid install
```

For some sample cases, like Linux systems based on the Intel(R) 64 architecture, the makefiles contain values that must be common. However, you need to be familiar with building an HPL and picking appropriate values for these variables.

New Features of Intel(R) Optimized MP LINPACK Benchmark

The toolset is basically identical with the HPL 2.0 distribution. There are a few changes that are optionally compiled in and disabled until you specifically request them. These new features are:

ASYOUGO: Provides non-intrusive performance information while runs proceed. There are only a few outputs and this information does not impact performance. This is especially useful because many runs can go for hours without any information.

ASYOUGO2: Provides slightly intrusive additional performance information by intercepting every `DGEMM` call.

ASYOUGO2_DISPLAY: Displays the performance of all the significant `DGEMMs` inside the run.

ENDEARLY: Displays a few performance hints and then terminates the run early.

FASTSWAP: Inserts the LAPACK-optimized `DLASWP` into HPL's code. You can experiment with this to determine best results.

HYBRID: Establishes the Hybrid OpenMP/MPI mode of MP LINPACK, providing the possibility to use threaded Intel(R) Math Kernel Library (Intel(R) MKL) and prebuilt MP LINPACK hybrid libraries.

CAUTION. Use this option only with an Intel compiler and the Intel (R) MPI library version 3.1 or higher. You are also recommended to use the compiler version 10.0 or higher.

Benchmarking a Cluster

To benchmark a cluster, follow the sequence of steps below (some of them are optional). Pay special attention to the iterative steps 3 and 4. They make a loop that searches for HPL parameters (specified in `HPL.dat`) that enable you to reach the top performance of your cluster.

1. Install HPL and make sure HPL is functional on all the nodes.
2. You may run `nodeperf.c` (included in the distribution) to see the performance of `DGEMM` on all the nodes.

Compile `nodeperf.c` with your MPI and Intel(R) Math Kernel Library (Intel(R) MKL). For example:

```
mpiicc -O3 nodeperf.c -L$MKLPATH $MKLPATH/libmkl_intel_lp64.a \
-Wl,--start-group $MKLPATH/libmkl_sequential.a \
$MKLPATH/libmkl_core.a -Wl,--end-group -lpthread .
```

Launching `nodeperf.c` on all the nodes is especially helpful in a very large cluster. `nodeperf` enables quick identification of the potential problem spot without numerous small MP LINPACK runs around the cluster in search of the bad node. It goes through all the nodes, one at a time, and reports the performance of `DGEMM` followed by some host identifier. Therefore, the higher the `DGEMM` performance, the faster that node was performing.

3. Edit `HPL.dat` to fit your cluster needs.
   Read through the HPL documentation for ideas on this. Note, however, that you should use at least 4 nodes.
4. Make an HPL run, using compile options such as `ASYOUGO`, `ASYOUGO2`, or `ENDEARLY` to aid in your search. These options enable you to gain insight into the performance sooner than HPL would normally give this insight.

   When doing so, follow these recommendations:

   - Use MP LINPACK, which is a patched version of HPL, to save time in the search.

     All performance intrusive features are compile-optional in MP LINPACK. That is, if you do not use the new options to reduce search time, these features are disabled. The primary purpose of the additions is to assist you in finding solutions.

     HPL requires a long time to search for many different parameters. In MP LINPACK, the goal is to get the best possible number.

     Given that the input is not fixed, there is a large parameter space you must search over. An exhaustive search of all possible inputs is improbably large even for a powerful cluster. MP LINPACK optionally prints information on performance as it proceeds. You can also terminate early.
   - Save time by compiling with `-DENDEARLY -DASYOUGO2` and using a negative threshold (do not use a negative threshold on the final run that you intend to submit as a Top500 entry). Set the threshold in line 13 of the HPL 2.0 input file `HPL.dat`
   - If you are going to run a problem to completion, do it with `-DASYOUGO`.
5. Using the quick performance feedback, return to step 3 and iterate until you are sure that the performance is as good as possible.

See Also

- Options to Reduce Search Time

Options to Reduce Search Time

Running large problems to completion on large numbers of nodes can take many hours. The search space for MP LINPACK is also large: not only can you run any size problem, but over a number of block sizes, grid layouts, lookahead steps, using different factorization methods, and so on. It can be a large waste of time to run a large problem to completion only to discover it ran 0.01% slower than your previous best problem.

Use the following options to reduce the search time:

- `-DASYOUGO`

- -DENDEARLY
- -DASYOUGO2

  Use -DASYOUGO2 cautiously because it does have a marginal performance impact. To see DGEMM internal performance, compile with -DASYOUGO2 and -DASYOUGO2_DISPLAY. These options provide a lot of useful DGEMM performance information at the cost of around 0.2% performance loss.

If you want to use the old HPL, simply omit these options and recompile from scratch. To do this, try "make arch=<*arch*> clean_arch_all".

-DASYOUGO

-DASYOUGO gives performance data as the run proceeds. The performance always starts off higher and then drops because this actually happens in LU decomposition (a decomposition of a matrix into a product of a lower (L) and upper (U) triangular matrices). The ASYOUGO performance estimate is usually an overestimate (because the LU decomposition slows down as it goes), but it gets more accurate as the problem proceeds. The greater the lookahead step, the less accurate the first number may be. ASYOUGO tries to estimate where one is in the LU decomposition that MP LINPACK performs and this is always an overestimate as compared to ASYOUGO2, which measures actually achieved DGEMM performance. Note that the ASYOUGO output is a subset of the information that ASYOUGO2 provides. So, refer to the description of the -DASYOUGO2 option below for the details of the output.

-DENDEARLY

-DENDEARLY t erminates the problem after a few steps, so that you can set up 10 or 20 HPL runs without monitoring them, see how they all do, and then only run the fastest ones to completion. -DENDEARLY assumes -DASYOUGO. You do not need to define both, although it doesn't hurt. To avoid the residual check for a problem that terminates early, set the "threshold" parameter in HPL.dat to a negative number when testing ENDEARLY. It also sometimes gives a better picture to compile with -DASYOUGO2 when using -DENDEARLY.

Usage notes on -DENDEARLY follow:

- -DENDEARLY stops the problem after a few iterations of DGEMM on the block size (the bigger the blocksize, the further it gets). It prints only 5 or 6 "updates", whereas -DASYOUGO prints about 46 or so output elements before the problem completes.
- Performance for -DASYOUGO and -DENDEARLY always starts off at one speed, slowly increases, and then slows down toward the end (because that is what LU does). -DENDEARLY is likely to terminate before it starts to slow down.
- -DENDEARLY terminates the problem early with an HPL Error exit. It means that you need to ignore the missing residual results, which are wrong because the problem never completed. However, you can get an idea what the initial performance was, and if it looks good, then run the problem to completion without -DENDEARLY. To avoid the error check, you can set HPL's threshold parameter in HPL.dat to a negative number.
- Though -DENDEARLY terminates early, HPL treats the problem as completed and computes Gflop rating as though the problem ran to completion. Ignore this erroneously high rating.
- The bigger the problem, the more accurately the last update that -DENDEARLY returns is close to what happens when the problem runs to completion. -DENDEARLY is a poor approximation for small problems. It is for this reason that you are suggested to use ENDEARLY in conjunction with ASYOUGO2, because ASYOUGO2 reports actual DGEMM performance, which can be a closer approximation to problems just starting.

-DASYOUGO2

-DASYOUGO2 gives detailed single-node DGEMM performance information. It captures all DGEMM calls (if you use Fortran BLAS) and records their data. Because of this, the routine has a marginal intrusive overhead. Unlike -DASYOUGO, which is quite non-intrusive, -DASYOUGO2 interrupts every DGEMM call to monitor its performance. You should beware of this overhead, although for big problems, it is, less than 0.1%.

Here is a sample ASYOUGO2 output (the first 3 non-intrusive numbers can be found in ASYOUGO and ENDEARLY), so it suffices to describe these numbers here:

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

The problem size was N=16000 with a block size of 128. After 10 blocks, that is, 1280 columns, an output was sent to the screen. Here, the fraction of columns completed is 1280/16000=0.08. Only up to 40 outputs are printed, at various places through the matrix decomposition: fractions

```
0.005 0.010 0.015 0.020 0.025 0.030 0.035 0.040 0.045 0.050 0.055 0.060 0.065 0.070 0.075
0.080 0.085 0.090 0.095 0.100 0.105 0.110 0.115 0.120 0.125 0.130 0.135 0.140 0.145 0.150
0.155 0.160 0.165 0.170 0.175 0.180 0.185 0.190 0.195 0.200 0.205 0.210 0.215 0.220 0.225
0.230 0.235 0.240 0.245 0.250 0.255 0.260 0.265 0.270 0.275 0.280 0.285 0.290 0.295 0.300
0.305 0.310 0.315 0.320 0.325 0.330 0.335 0.340 0.345 0.350 0.355 0.360 0.365 0.370 0.375
0.380 0.385 0.390 0.395 0.400 0.405 0.410 0.415 0.420 0.425 0.430 0.435 0.440 0.445 0.450
0.455 0.460 0.465 0.470 0.475 0.480 0.485 0.490 0.495 0.515 0.535 0.555 0.575 0.595 0.615
0.635 0.655 0.675 0.695 0.795 0.895.
```

However, this problem size is so small and the block size so big by comparison that as soon as it prints the value for 0.045, it was already through 0.08 fraction of the columns. On a really big problem, the fractional number will be more accurate. It never prints more than the 112 numbers above. So, smaller problems will have fewer than 112 updates, and the biggest problems will have precisely 112 updates.

Mflops is an estimate based on 1280 columns of LU being completed. However, with lookahead steps, sometimes that work is not actually completed when the output is made. Nevertheless, this is a good estimate for comparing identical runs.

The 3 numbers in parenthesis are intrusive ASYOUGO2 addins. DT is the total time processor 0 has spent in DGEMM. DF is the number of billion operations that have been performed in DGEMM by one processor. Hence, the performance of processor 0 (in Gflops) in DGEMM is always DF/DT. Using the number of DGEMM flops as a basis instead of the number of LU flops, you get a lower bound on performance of the run by looking at DMF, which can be compared to Mflops above (It uses the global LU time, but the DGEMM flops are computed under the assumption that the problem is evenly distributed amongst the nodes, as only HPL's node (0,0) returns any output.)

Note that when using the above performance monitoring tools to compare different HPL.dat input data sets, you should be aware that the pattern of performance drop-off that LU experiences is sensitive to some input data. For instance, when you try very small problems, the performance drop-off from the initial values to end values is very rapid. The larger the problem, the less the drop-off, and it is probably safe to use the first few performance values to estimate the difference between a problem size 700000 and 701000, for instance. Another factor that influences the performance drop-off is the grid dimensions (P and Q). For big problems, the performance tends to fall off less from the first few steps when P and Q are roughly equal in value. You can make use of a large number of parameters, such as broadcast types, and change them so that the final performance is determined very closely by the first few steps.

Using these tools will greatly assist the amount of data you can test.

See Also

- Benchmarking a Cluster

Intel(R) Math Kernel Library Language Interfaces Support

A

Language Interfaces Support, by Function Domain

The following table shows language interfaces that Intel(R) Math Kernel Library (Intel(R) MKL) provides for each function domain. However, Intel MKL routines can be called from other languages using mixed-language programming. See Mixed-language Programming with Intel(R) MKL for an example of how to call Fortran routines from C/C++.

Function Domain	FORTRAN 77 interface	Fortran 90/95 interface	C/C++ interface
Basic Linear Algebra Subprograms (BLAS)	Yes	Yes	via CBLAS
BLAS-like extension transposition routines	Yes		Yes
Sparse BLAS Level 1	Yes	Yes	via CBLAS
Sparse BLAS Level 2 and 3	Yes	Yes	Yes
LAPACK routines for solving systems of linear equations	Yes	Yes	Yes
LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations	Yes	Yes	Yes
Auxiliary and utility LAPACK routines	Yes		Yes
Parallel Basic Linear Algebra Subprograms (PBLAS)	Yes		
ScaLAPACK routines	Yes		†
DSS/PARDISO* solvers	Yes	Yes	Yes
Other Direct and Iterative Sparse Solver routines	Yes	Yes	Yes
Vector Mathematical Library (VML) functions	Yes	Yes	Yes
Vector Statistical Library (VSL) functions	Yes	Yes	Yes
Fourier Transform functions (FFT)		Yes	Yes
Cluster FFT functions		Yes	Yes

Function Domain	FORTRAN 77 interface	Fortran 90/95 interface	C/C++ interface
Trigonometric Transform routines		Yes	Yes
Fast Poisson, Laplace, and Helmholtz Solver (Poisson Library) routines		Yes	Yes
Optimization (Trust-Region) Solver routines	Yes	Yes	Yes
GMP* arithmetic functions			Yes
Support functions (including memory allocation)	Yes	Yes	Yes

† Supported using a mixed language programming call. See Intel(R) MKL Include Files for the respective header file.

Include Files

Function domain	Fortran Include Files	C/C++ Include Files
All function domains	`mkl.fi`	`mkl.h`
BLAS Routines	`blas.f90` `mkl_blas.fi`	`mkl_blas.h`
BLAS-like Extension Transposition Routines	`mkl_trans.fi`	`mkl_trans.h`
CBLAS Interface to BLAS		`mkl_cblas.h`
Sparse BLAS Routines	`mkl_spblas.fi`	`mkl_spblas.h`
LAPACK Routines	`lapack.f90` `mkl_lapack.fi`	`mkl_lapack.h`
C Interface to LAPACK		`mkl_lapacke.h`
ScaLAPACK Routines		`mkl_scalapack.h`
All Sparse Solver Routines	`mkl_solver.f90`	`mkl_solver.h`
PARDISO	`mkl_pardiso.f77` `mkl_pardiso.f90`	`mkl_pardiso.h`
DSS Interface	`mkl_dss.f77` `mkl_dss.f90`	`mkl_dss.h`
RCI Iterative Solvers ILU Factorization	`mkl_rci.fi`	`mkl_rci.h`
Optimization Solver Routines	`mkl_rci.fi`	`mkl_rci.h`

Function domain	Fortran Include Files	C/C++ Include Files
Vector Mathematical Functions	`mkl_vml.f77` `mkl_vml.90`	`mkl_vml.h`
Vector Statistical Functions	`mkl_vsl.f77` `mkl_vsl.f90`	`mkl_vsl_functions.h`
Fourier Transform Functions	`mkl_dfti.f90`	`mkl_dfti.h`
Cluster Fourier Transform Functions	`mkl_cdft.f90`	`mkl_cdft.h`
Partial Differential Equations Support Routines		
Trigonometric Transforms	`mkl_trig_transforms.f90`	`mkl_trig_transforms.h`
Poisson Solvers	`mkl_poisson.f90`	`mkl_poisson.h`
GMP interface		`mkl_gmp.h`
Support functions	`mkl_service.f90` `mkl_service.fi`	`mkl_service.h`
Memory allocation routines		`i_malloc.h`
MKL examples interface		`mkl_example.h`

See Also

- Language Interfaces Support, by Function Domain

Support for Third-Party Interfaces

B

GMP* Functions

Intel(R) Math Kernel Library (Intel(R) MKL) implementation of GMP* arithmetic functions includes arbitrary precision arithmetic operations on integer numbers. The interfaces of such functions fully match the GNU Multiple Precision* (GMP) Arithmetic Library. For specifications of these functions, please see http://www.intel.com/software/products/mkl/docs/gnump/WebHelp/.

If you currently use the GMP* library, you need to modify `INCLUDE` statements in your programs to `mkl_gmp.h`.

FFTW Interface Support

Intel(R) Math Kernel Library (Intel(R) MKL) offers two collections of wrappers for the FFTW interface (www.fftw.org). The wrappers are the superstructure of FFTW to be used for calling the Intel MKL Fourier transform functions. These collections correspond to the FFTW versions 2.x and 3.x and the Intel MKL versions 7.0 and later.

These wrappers enable using Intel MKL Fourier transforms to improve the performance of programs that use FFTW, without changing the program source code. See the "*FFTW Interface to Intel(R) Math Kernel Library*" appendix in the Intel MKL Reference Manual for details on the use of the wrappers.

Directory Structure in Detail

C

Tables in this section show contents of the Intel(R) Math Kernel Library (Intel(R) MKL) architecture-specific directories.

Optimization Notice
The Intel® Math Kernel Library (Intel® MKL) contains functions that are more highly optimized for Intel microprocessors than for other microprocessors. While the functions in Intel® MKL offer optimizations for both Intel and Intel-compatible microprocessors, depending on your code and other factors, you will likely get extra performance on Intel microprocessors. While the paragraph above describes the basic optimization approach for Intel® MKL as a whole, the library may or may not be optimized to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Intel recommends that you evaluate other library products to determine which best meets your requirements.

Detailed Structure of the IA-32 Architecture Directories

Static Libraries in the IA-32 Architecture Directory `lib/ia32`

File	Contents
Interface layer	
`libmkl_intel.a`	Interface library for the Intel(R) compilers
`libmkl_blas95.a`	Fortran 95 interface library for BLAS for the Intel(R) Fortran compiler
`libmkl_lapack95.a`	Fortran 95 interface library for LAPACK for the Intel Fortran compiler
`libmkl_gf.a`	Interface library for the GNU* Fortran compiler
Threading layer	
`libmkl_intel_thread.a`	Threading library for the Intel compilers

File	Contents
libmkl_gnu_thread.a	Threading library for the GNU Fortran and C compilers
libmkl_pgi_thread.a	Threading library for the PGI* compiler
libmkl_sequential.a	Sequential library
Computational layer	
libmkl_core.a	Kernel library for the IA-32 architecture
libmkl_solver.a	Deprecated. Empty library for backward compatibility
libmkl_solver_sequential.a	Deprecated. Empty library for backward compatibility
libmkl_scalapack_core.a	ScaLAPACK routines
libmkl_cdft_core.a	Cluster version of FFT functions
Run-time Libraries (RTL)	
libmkl_blacs.a	BLACS routines supporting the following MPICH versions: • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2
libmkl_blacs_intelmpi.a	BLACS routines supporting Intel MPI and MPICH2
libmkl_blacs_intelmpi20.a	A soft link to `lib/32/libmkl_blacs_intelmpi.a`
libmkl_blacs_openmpi.a	BLACS routines supporting OpenMPI

Dynamic Libraries in the IA-32 Architecture Directory `lib/ia32`

File	Contents
Interface layer	
libmkl_rt.so	Single Dynamic Library interface library
libmkl_intel.so	Interface library for the Intel compilers
libmkl_gf.so	Interface library for the GNU Fortran compiler
Threading layer	
libmkl_intel_thread.so	Threading library for the Intel compilers
libmkl_gnu_thread.so	Threading library for the GNU Fortran and C compilers

File	Contents
libmkl_pgi_thread.so	Threading library for the PGI* compiler
libmkl_sequential.so	Sequential library
Computational layer	
libmkl_core.so	Library dispatcher for dynamic load of processor-specific kernel library
libmkl_def.so	Default kernel library (Intel(R) Pentium(R), Pentium(R) Pro, Pentium(R) II, and Pentium(R) III processors)
libmkl_p4.so	Pentium(R) 4 processor kernel library
libmkl_p4p.so	Kernel library for the Intel(R) Pentium(R) 4 processor with Streaming SIMD Extensions 3 (SSE3), including Intel(R) Core(TM) Duo and Intel(R) Core(TM) Solo processors.
libmkl_p4m.so	Kernel library for processors based on the Intel(R) Core(TM) microarchitecture (except Intel(R) Core(TM) Duo and Intel(R) Core(TM) Solo processors, for which mkl_p4p.so is intended)
libmkl_p4m3.so	Kernel library for the Intel(R) Core(TM) i7 processors
libmkl_vml_def.so	VML/VSL part of default kernel for old Intel(R) Pentium(R) processors
libmkl_vml_ia.so	VML/VSL default kernel for newer Intel(R) architecture processors
libmkl_vml_p4.so	VML/VSL part of Pentium(R) 4 processor kernel
libmkl_vml_p4m.so	VML/VSL for processors based on the Intel(R) Core(TM) microarchitecture
libmkl_vml_p4m2.so	VML/VSL for 45nm Hi-k Intel(R) Core(TM)2 and Intel Xeon(R) processor families
libmkl_vml_p4m3.so	VML/VSL for the Intel(R) Core(TM) i7 processors
libmkl_vml_p4p.so	VML/VSL for Pentium(R) 4 processor with Streaming SIMD Extensions 3 (SSE3)
libmkl_vml_avx.so	VML/VSL optimized for the Intel(R) Advanced Vector Extensions (Intel(R) AVX)
libmkl_scalapack_core.so	ScaLAPACK routines.
libmkl_cdft_core.so	Cluster version of FFT functions.
Run-time Libraries (RTL)	
libmkl_blacs_intelmpi.so	BLACS routines supporting Intel MPI and MPICH2
locale/en_US/mkl_msg.cat	Catalog of Intel(R) Math Kernel Library (Intel(R) MKL) messages in English
locale/ja_JP/mkl_msg.cat	Catalog of Intel MKL messages in Japanese. Available only if the Intel(R) MKL package provides Japanese localization. Please see the Release Notes for this information

Detailed Structure of the Intel(R) 64 Architecture Directories

Static Libraries in the `lib/intel64` Directory

File	Contents
Interface layer	
`libmkl_intel_lp64.a`	LP64 interface library for the Intel compilers
`libmkl_intel_ilp64.a`	ILP64 interface library for the Intel compilers
`libmkl_intel_sp2dp.a`	SP2DP interface library for the Intel compilers
`libmkl_blas95_lp64.a`	Fortran 95 interface library for BLAS for the Intel(R) Fortran compiler. Supports the LP64 interface
`libmkl_blas95_ilp64.a`	Fortran 95 interface library for BLAS for the Intel(R) Fortran compiler. Supports the ILP64 interface
`libmkl_lapack95_lp64.a`	Fortran 95 interface library for LAPACK for the Intel(R) Fortran compiler. Supports the LP64 interface
`libmkl_lapack95_ilp64.a`	Fortran 95 interface library for LAPACK for the Intel(R) Fortran compiler. Supports the ILP64 interface
`libmkl_gf_lp64.a`	LP64 interface library for the GNU Fortran compilers
`libmkl_gf_ilp64.a`	ILP64 interface library for the GNU Fortran compilers
Threading layer	
`libmkl_intel_thread.a`	Threading library for the Intel compilers
`libmkl_gnu_thread.a`	Threading library for the GNU Fortran and C compilers
`libmkl_pgi_thread.a`	Threading library for the PGI compiler
`libmkl_sequential.a`	Sequential library
Computational layer	
`libmkl_core.a`	Kernel library for the Intel(R) 64 architecture
`libmkl_solver_lp64.a`	Deprecated. Empty library for backward compatibility
`libmkl_solver_lp64_sequential.a`	Deprecated. Empty library for backward compatibility
`libmkl_solver_ilp64.a`	Deprecated. Empty library for backward compatibility
`libmkl_solver_ilp64_sequential.a`	Deprecated. Empty library for backward compatibility

File	Contents
libmkl_scalapack_lp64.a	ScaLAPACK routine library supporting the LP64 interface
libmkl_scalapack_ilp64.a	ScaLAPACK routine library supporting the ILP64 interface
libmkl_cdft_core.a	Cluster version of FFT functions.
Run-time Libraries (RTL)	
libmkl_blacs_lp64.a	LP64 version of BLACS routines supporting the following MPICH versions: • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2
libmkl_blacs_ilp64.a	ILP64 version of BLACS routines supporting the following MPICH versions: • Myricom* MPICH version 1.2.5.10 • ANL* MPICH version 1.2.5.2
libmkl_blacs_intelmpi_lp64.a	LP64 version of BLACS routines supporting Intel MPI and MPICH2
libmkl_blacs_intelmpi_ilp64.a	ILP64 version of BLACS routines supporting Intel MPI and MPICH2
libmkl_blacs_intelmpi20_lp64.a	A soft link to lib/intel64/libmkl_blacs_intelmpi_lp64.a
libmkl_blacs_intelmpi20_ilp64.a	A soft link to lib/intel64/libmkl_blacs_intelmpi_ilp64.a
libmkl_blacs_openmpi_lp64.a	LP64 version of BLACS routines supporting OpenMPI.
libmkl_blacs_openmpi_ilp64.a	ILP64 version of BLACS routines supporting OpenMPI.
libmkl_blacs_sgimpt_lp64.a	LP64 version of BLACS routines supporting SGI MPT.
libmkl_blacs_sgimpt_ilp64.a	ILP64 version of BLACS routines supporting SGI MPT.

Dynamic Libraries in the Intel(R) 64 Architecture Directory lib/intel64

File	Contents
Interface layer	
libmkl_rt.so	Single Dynamic Library interface library
libmkl_intel_lp64.so	LP64 interface library for the Intel compilers
libmkl_intel_ilp64.so	ILP64 interface library for the Intel compilers
libmkl_intel_sp2dp.so	SP2DP interface library for the Intel compilers

File	Contents
libmkl_gf_lp64.so	LP64 interface library for the GNU Fortran compilers
libmkl_gf_ilp64.so	ILP64 interface library for the GNU Fortran compilers
Threading layer	
libmkl_intel_thread.so	Threading library for the Intel compilers
libmkl_gnu_thread.so	Threading library for the GNU Fortran and C compilers
libmkl_pgi_thread.so	Threading library for the PGI* compiler
libmkl_sequential.so	Sequential library
Computational layer	
libmkl_core.so	Library dispatcher for dynamic load of processor-specific kernel
libmkl_def.so	Default kernel library
libmkl_mc.so	Kernel library for processors based on the Intel(R) Core(TM) microarchitecture
libmkl_mc3.so	Kernel library for the Intel(R) Core(TM) i7 processors
libmkl_avx.so	Kernel optimized for the Intel(R) Advanced Vector Extensions (Intel(R) AVX).
libmkl_vml_def.so	VML/VSL part of default kernels
libmkl_vml_p4n.so	VML/VSL for the Intel(R) Xeon(R) processor using the Intel(R) 64 architecture
libmkl_vml_mc.so	VML/VSL for processors based on the Intel(R) Core(TM) microarchitecture
libmkl_vml_mc2.so	VML/VSL for 45nm Hi-k Intel(R) Core(TM)2 and Intel Xeon(R) processor families
libmkl_vml_mc3.so	VML/VSL for the Intel(R) Core(TM) i7 processors
libmkl_vml_avx.so	VML/VSL optimized for the Intel(R) Advanced Vector Extensions (Intel(R) AVX)
libmkl_scalapack_lp64.so	ScaLAPACK routine library supporting the LP64 interface
libmkl_scalapack_ilp64.so	ScaLAPACK routine library supporting the ILP64 interface
libmkl_cdft_core.so	Cluster version of FFT functions.
Run-time Libraries (RTL)	
libmkl_intelmpi_lp64.so	LP64 version of BLACS routines supporting Intel MPI and MPICH2
libmkl_intelmpi_ilp64.so	ILP64 version of BLACS routines supporting Intel MPI and MPICH2
locale/en_US/mkl_msg.cat	Catalog of Intel(R) Math Kernel Library (Intel(R) MKL) messages in English

File	Contents
`locale/ja_JP/mkl_msg.cat`	Catalog of Intel MKL messages in Japanese. Available only if the Intel(R) MKL package provides Japanese localization. Please see the Release Notes for this information

Index

I

J

L

M

N

P

S

T

U