Intel® Math Kernel Library for Linux* OS

User's Guide

Intel® MKL - Linux OS*

Document Number: 314774-028US

Legal Information

Contents

Legal Information

Introducing the Intel® Math Kernel Library

Intel® Math Kernel Library (Intel® MKL) is a computing math library of highly optimized, extensively threaded routines for applications that require maximum performance. Intel MKL provides comprehensive functionality support in these major areas of computation:

- BLAS (level 1, 2, and 3) and LAPACK linear algebra routines, offering vector, vector-matrix, and matrix-matrix operations.
- The PARDISO* direct sparse solver, an iterative sparse solver, and supporting sparse BLAS (level 1, 2, and 3) routines for solving sparse systems of equations.
- ScaLAPACK distributed processing linear algebra routines for Linux* and Windows* operating systems, as well as the Basic Linear Algebra Communications Subprograms (BLACS) and the Parallel Basic Linear Algebra Subprograms (PBLAS).
- Fast Fourier transform (FFT) functions in one, two, or three dimensions with support for mixed radices (not limited to sizes that are powers of 2), as well as distributed versions of these functions provided for use on clusters of the Linux* and Windows* operating systems.
- Vector Math Library (VML) routines for optimized mathematical operations on vectors.
- Vector Statistical Library (VSL) routines, which offer high-performance vectorized random number generators (RNG) for several probability distributions, convolution and correlation routines, and summary statistics functions.
- Data Fitting Library, which provides capabilities for spline-based approximation of functions, derivatives and integrals of functions, and search.

For details see the *Intel® MKL Reference Manual*.

Intel MKL is optimized for the latest Intel processors, including processors with multiple cores (see the *Intel® MKL Release Notes* for the full list of supported processors). Intel MKL also performs well on non-Intel processors.

For Linux* systems based on Intel® 64 Architecture, Intel MKL also includes support for the Intel® Many Integrated Core (Intel® MIC) Architecture and provides libraries to help you port your applications to Intel MIC Architecture.

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 MKL support website at http://www.intel.com/software/products/support/.

The Intel MKL 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 MKL directories.

`<parent product directory>`	The installation directory for the larger product that includes Intel MKL; for example, Intel® C++ Composer XE or Intel® Fortran Composer XE.
`<mkl directory>`	The main directory where Intel MKL is installed: *`<mkl directory>`*`=`*`<parent product directory>`*`/mkl`. Replace this placeholder with the specific pathname in the configuring, linking, and building instructions.

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: `./benchmarks/linpack`
`Monospace lowercase mixed with uppercase`	Indicates: • Commands and command-line options, for example, `icc myprog.c -L$MKLPATH -I$MKLINCLUDE -lmkl -liomp5 -lpthread` • Filenames, directory names, and pathnames, for example, • 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, for example, *`lda`*. 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.

Overview 1

Document Overview

The Intel® Math Kernel Library (Intel® 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. The document 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

Language Interfaces Support, by Function Domain

What's New

This User's Guide documents Intel® Math Kernel Library (Intel® MKL) 11.0 update 1.

The following new functionality and features of the product have been described in the document:

- A low-communication algorithm for Cluster FFT (see Enabling low-communication algorithm in Cluster FFT).
- Linking with Intel MKL cluster software on Intel® Xeon Phi™ coprocessors in native mode (see Linking with ScaLAPACK and Cluster FFTs on Intel® Xeon Phi™ Coprocessors).

Additionally, minor updates have been made to correct errors in the document.

Related Information

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

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

Getting Started

2

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Checking Your Installation

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

1. Intel MKL installs in the `<parent product directory>` directory.

   Check that the subdirectory of `<parent product directory>` referred to as `<mkl directory>` was created.
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, as explained in Setting Environment Variables
4. To understand how the Intel MKL directories are structured, see Intel® Math Kernel Library Structure.
5. To make sure that Intel MKL runs on your system, launch an Intel MKL example, as explained in Using Code Examples.

See Also

Notational Conventions

Setting Environment Variables

See Also

Setting the Number of Threads Using an OpenMP* Environment Variable

Scripts to Set Environment Variables

When the installation of Intel MKL for Linux* OS is complete, set the `INCLUDE`, `MKLROOT`, `LD_LIBRARY_PATH`, `MIC_LD_LIBRARY_PATH`, `MANPATH`, `LIBRARY_PATH`, `CPATH`, and `NLSPATH` environment variables in the command shell using one of the script files in the `bin` subdirectory of the Intel MKL installation directory. The environment variable `MIC_LD_LIBRARY_PATH` specifies locations of shared objects for Intel® MIC Architecture.

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® 64	C	`intel64/mklvars_intel64.csh`
Intel® 64	Bash and Bourne (sh)	`intel64/mklvars_intel64.sh`
IA-32 and Intel® 64	C	`mklvars.csh`
IA-32 and Intel® 64	Bash and Bourne (sh)	`mklvars.sh`

Running the Scripts

The parameters of the scripts specify the following:

- Architecture.
- Use of Intel MKL Fortran modules precompiled with the Intel® Fortran compiler. Supply this parameter only if you are using this compiler.
- Programming interface (LP64 or ILP64).

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

Script	Architecture (required, when applicable)	Use of Fortran 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.sh ia32`
  sets the environment for Intel MKL to use the IA-32 architecture.
- The command
  `mklvars.sh intel64 mod ilp64`
  sets the environment for Intel MKL to use the Intel® 64 architecture, ILP64 programming interface, and Fortran modules.
- The command

```
mklvars.sh intel64 mod
```

sets the environment for Intel MKL to use the Intel® 64 architecture, LP64 interface, and Fortran modules.

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
Interface Libraries and Modules
Fortran 95 Interfaces to LAPACK and BLAS
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`, 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 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]`

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 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

Include Files

Using Code Examples

The Intel 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 instance, 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. You can find examples of Automatic Offload in the `examples/mic_ao` subdirectory and examples of Compiler Assisted Offload in `examples/mic_offload` subdirectory. Source code for the examples is in the next-level `sources` subdirectory.

See Also

High-level Directory Structure

Using Intel® Math Kernel Library on Intel® Xeon Phi™ Coprocessors

What You Need to Know Before You Begin Using the Intel® Math Kernel Library

Target platform	Identify the architecture of your target machine: • IA-32 or compatible • Intel® 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
- Data Fitting 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 Other 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® 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® Math Kernel Library.

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 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 Improving Performance with Threading.

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® Math Kernel Library.

MPI used

Decide what MPI you will use with the Intel MKL cluster software. You are strongly encouraged to use the latest available version of Intel® MPI.

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® Math Kernel Library

3

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Architecture Support

Intel® Math Kernel Library (Intel® MKL) for Linux* OS provides architecture-specific implementations for supported platforms. 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® 64 or compatible	`<mkl directory>/lib/intel64`
Intel® Many Integrated Core (Intel® MIC)	`<mkl directory>/lib/mic`

See Also

High-level Directory Structure
Notational Conventions
Detailed Structure of the IA-32 Architecture Directories
Detailed Structure of the Intel® 64 Architecture Directories

High-level Directory Structure

Directory	Contents
`<mkl directory>`	Installation directory of the Intel® Math Kernel Library (Intel® MKL)
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® 64 architecture
`benchmarks/linpack`	Shared-memory (SMP) version of the LINPACK benchmark
`benchmarks/mp_linpack`	Message-passing interface (MPI) version of the LINPACK benchmark

Directory	Contents
`examples`	Examples directory. Each subdirectory has source and data files
`examples/mic_ao`	Examples of automatic offloading Intel MKL computations to Intel® Xeon Phi™ coprocessors. Each subdirectory has source files.
`examples/mic_offload`	Examples of Compiler Assisted Offload of computations to Intel Xeon Phi coprocessors. Each subdirectory has source 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® Fortran compiler
`include/intel64/lp64`	Fortran 95 .mod files for the Intel® 64 architecture, Intel Fortran compiler, and LP64 interface
`include/intel64/ilp64`	Fortran 95 .mod files for the Intel® 64 architecture, Intel Fortran compiler, and ILP64 interface
`include/fftw`	Header files for the FFTW2 and FFTW3 interfaces
`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® 64 architecture
`lib/mic`	Static libraries and shared objects for the Intel® MIC architecture
`tests`	Source and data files for tests
`tools`	Tools and plug-ins
`tools/builder`	Tools for creating custom dynamically linkable libraries
Subdirectories of *`<parent product directory>`*	
`Documentation/en_US/mkl`	Intel MKL documentation.

See Also

Notational Conventions

Layered Model Concept

Intel 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 run-time libraries (RTL).

The following table provides more details of each layer.

Layer	**Description**

Interface Layer	This 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).
	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	This layer is the heart of Intel MKL. It 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 Run-time Libraries (RTL)	To support threading with Intel® compilers, Intel MKL uses RTLs of the Intel® C++ Compiler or Intel® Fortran Compiler. 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® Math Kernel Library
Linking with Threading Libraries

Contents of the Documentation Directories

Most of Intel MKL documentation is installed in `<parent product directory>/Documentation/<locale>/mkl`. For example, the documentation in English is installed in `<parent product directory>/Documentation/en_US/mkl`. However, some Intel MKL-related documents are installed one or two levels up. The following table lists Intel MKL-related documentation.

File name	Comment
Files in `<parent product directory>/Documentation`	
`<locale>/clicense` or `<locale>/flicense`	Common end user license for the larger product that includes Intel MKL.
`mklsupport.txt`	Information on package number for customer support reference
Contents of `<parent product directory>/Documentation/<locale>/mkl`	
`redist.txt`	List of redistributable files
`mkl_documentation.htm`	Overview and links for the Intel MKL documentation
`get_started.html`	Brief introduction to Intel MKL
`tutorials/mkl_mmx_c/index.htm`	Getting Started Tutorials "Using Intel® Math Kernel Library for Matrix Multiplication (C Language)"
`tutorials/mkl_mmx_f/index.htm`	Getting Started Tutorials "Using Intel® Math Kernel Library for Matrix Multiplication (Fortran Language)"
`mklman90.pdf` †	Intel MKL 9.0 Reference Manual in Japanese
`Release_Notes.htm`	Intel MKL Release Notes
`mkl_userguide/index.htm`	Intel MKL User's Guide in an uncompressed HTML format, this document
`mkl_link_line_advisor.htm`	Intel MKL Link-line Advisor

† Included only in the Japanese versions of the Intel® C++ Composer XE and Intel® Fortran Composer XE.

For more documents, search Intel MKL documentation at http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/.

See Also

Notational Conventions

Linking Your Application with the Intel® Math Kernel Library

4

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Linking Quick Start

Intel® Math Kernel Library (Intel® MKL) provides several options for quick linking of your application, which depend on the way you link:

Using the Intel® Composer XE compiler	see Using the `-mkl` Compiler Option.
Explicit dynamic linking	see Using the Single Dynamic Library for how to simplify your link line.
Explicitly listing libraries on your link line	see Selecting Libraries to Link with for a summary of the libraries.
Using an interactive interface	see Using the Link-line Advisor to determine libraries and options to specify on your link or compilation line.
Using an internally provided tool	see Using the Command-line Link Tool to determine libraries, options, and environment variables or even compile and build your application.

Using the -mkl Compiler Option

The Intel® Composer XE compiler supports the following variants of the `-mkl` compiler option:

`-mkl` or `-mkl=parallel`	to link with standard threaded Intel MKL.
`-mkl=sequential`	to link with sequential version of Intel MKL.
`-mkl=cluster`	to link with Intel MKL cluster components (sequential) that use Intel MPI.

For more information on the `-mkl` compiler option, see the Intel Compiler User and Reference Guides.

On Intel® 64 architecture systems, for each variant of the `-mkl` option, the compiler links your application using the LP64 interface.

If you specify any variant of the `-mkl` compiler option, the compiler automatically includes the Intel MKL libraries. In cases not covered by the option, use the Link-line Advisor or see Linking in Detail.

See Also

Listing Libraries on a Link Line
Using the ILP64 Interface vs. LP64 Interface
Using the Link-line Advisor
Intel® Software Documentation Library for Intel® Composer XE documentation

Using the Single Dynamic Library

You can simplify your link line through the use of the Intel MKL Single Dynamic Library (SDL).

To use SDL, place `libmkl_rt.so` on your link line. For example:

```
icc application.c -lmkl_rt
```

SDL enables you to select the interface and threading library for Intel MKL at run time. By default, linking with SDL provides:

- LP64 interface on systems based on the Intel® 64 architecture
- Intel threading

To use other interfaces or change threading preferences, including use of the sequential version of Intel MKL, you need to specify your choices using functions or environment variables as explained in section Dynamically Selecting the Interface and Threading Layer.

Selecting Libraries to Link with

To link with Intel MKL:

- Choose one library from the Interface layer and one library from the Threading layer
- Add the only library from the Computational layer and run-time libraries (RTL)

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_intel.so`	`libmkl_intel_thread.so`	`libmkl_core.so`	`libiomp5.so`
Intel® 64 architecture, static linking	`libmkl_intel_lp64.a`	`libmkl_intel_thread.a`	`libmkl_core.a`	`libiomp5.so`
Intel® 64 architecture, dynamic linking	`libmkl_intel_lp64.so`	`libmkl_intel_thread.so`	`libmkl_core.so`	`libiomp5.so`
Intel® Many Integrated Core (Intel® MIC) Architecture, static linking	`libmkl_intel_lp64.a`	`libmkl_intel_thread.a`	`libmkl_core.a`	`libiomp5.so`

	Interface layer	Threading layer	Computational layer	RTL
Intel MIC Architecture, dynamic linking	`libmkl_intel_lp64.so`	`libmkl_intel_thread.so`	`libmkl_core.so`	`libiomp5.so`

The Single Dynamic Library (SDL) automatically links interface, threading, and computational libraries and thus simplifies linking. The following table lists Intel MKL libraries for dynamic linking using SDL. See Dynamically Selecting the Interface and Threading Layer for how to set the interface and threading layers at run time through function calls or environment settings.

	SDL	RTL
IA-32 and Intel® 64 architectures	`libmkl_rt.so`	`libiomp5.so`[††]

[††] Use the Link-line Advisor to check whether you need to explicitly link the `libiomp5.so` RTL.

For exceptions and alternatives to the libraries listed above, see Linking in Detail.

See Also

Layered Model Concept
Using the Link-line Advisor
Using the -mkl Compiler Option
Working with the Intel® Math Kernel Library Cluster Software
Linking on Intel Xeon Phi CoprocessorsFor special linking needed for Compiler Assisted Offload

Using the Link-line Advisor

Use the Intel MKL Link-line Advisor to determine the libraries and options to specify on your link or compilation line.

The latest version of the tool is available at http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor. The tool is also available in the product.

The 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

Contents of the Documentation Directoriesfor the location of the installed Link-line Advisor

Using the Command-line Link Tool

Use the command-line Link tool provided by Intel MKL to simplify building your application with Intel MKL.

The tool not only provides the options, libraries, and environment variables to use, but also performs compilation and building of your application.

The tool `mkl_link_tool` is installed in the `<mkl directory>/tools` directory.

See the knowledge base article at http://software.intel.com/en-us/articles/mkl-command-line-link-tool for more information.

Linking Examples

See Also

Using the Link-line 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 `ifort` with `icc`.

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

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.

- Static linking of `myprog.f` and parallel Intel 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 -lm
```

- 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 -lm
```

- 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 -lm
```

- 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 -lm
```

- 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_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 -lm
```

- 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 -lm
```

See Also

Fortran 95 Interfaces to LAPACK and BLAS
Examples for Linking a C Application
Examples for Linking a Fortran Application

Using the Single Dynamic Library
Linking with System Librariesfor specifics of linking with a GNU or PGI compiler

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 `ifort` with `icc`.

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

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.

- Static linking of `myprog.f` and parallel Intel 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 -lm
```

- 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 -lm
```

- 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 -lm
```

- 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 -lm
```

- 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 -lm
```

- 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 -lm
```

- 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 -lm
```

- 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 -lm
```

See Also

Fortran 95 Interfaces to LAPACK and BLAS
Examples for Linking a C Application
Examples for Linking a Fortran Application
Using the Single Dynamic Library
Linking with System Librariesfor specifics of linking with a GNU or PGI compiler

Linking in Detail

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

Listing Libraries on a Link Line

To link with Intel MKL, 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] [-ldl]
```

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 Link-line Advisor
Linking Examples
Working with the Intel® Math Kernel Library Cluster Software

Dynamically Selecting the Interface and Threading Layer

The Single Dynamic Library (SDL) enables you to dynamically select the interface and threading layer for Intel MKL.

Setting the Interface Layer

Available interfaces depend on the architecture of your system.

On systems based on the Intel® 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_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.

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 the 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.

See Also

Using the Single Dynamic Library
Layered Model Concept
Directory Structure in Detail

Linking with Interface Libraries

Using the ILP64 Interface vs. LP64 Interface

The Intel 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`

Integer Types	Fortran	C or C++
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`

To determine the type of an integer parameter of a function, use appropriate include files. For functions that support only a Fortran interface, use the C/C++ include files `*.h`.

The above table explains which integer parameters of functions become 64-bit and which remain 32-bit for ILP64. The table applies to most Intel MKL functions except some VML and VSL functions, which require integer parameters to be 64-bit or 32-bit regardless of the interface:

- **VML:** The *`mode`* parameter of VML functions is 64-bit.
- **Random Number Generators (RNG):**

  All discrete RNG except `viRngUniformBits64` are 32-bit.

  The `viRngUniformBits64` generator function and `vslSkipAheadStream` service function are 64-bit.
- **Summary Statistics:** The *`estimate`* parameter of the `vslsSSCompute/vsldSSCompute` function is 64-bit.

Refer to the *Intel MKL Reference Manual* for more information.

To better understand ILP64 interface details, see also examples and tests.

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`.

See Also

High-level Directory Structure
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 MKL package, they are prebuilt for the Intel® 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 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
Improving Performance with Threading
Avoiding Conflicts in the Execution Environment
Linking Examples

Selecting the Threading Layer

Several compilers that Intel 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 more 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	

Compiler	Application Threaded?	Threading Layer	RTL Recommended	Comment
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 MKL cluster software, you need to link your application with only one computational library, depending on the linking method:

Static Linking	Dynamic Linking
`libmkl_core.a`	`libmkl_core.so`

Computational Libraries for Applications that Use the Intel MKL Cluster Software

ScaLAPACK and Cluster Fourier Transform Functions (Cluster FFTs) require more computational libraries, which may depend on your architecture.

The following table lists computational libraries for IA-32 architecture applications that use ScaLAPACK or Cluster FFTs.

Computational Libraries for IA-32 Architecture

Function domain	Static Linking	Dynamic Linking
ScaLAPACK †	`libmkl_scalapack_core.a` `libmkl_core.a`	`libmkl_scalapack_core.so` `libmkl_core.so`
Cluster Fourier Transform Functions†	`libmkl_cdft_core.a` `libmkl_core.a`	`libmkl_cdft_core.so` `libmkl_core.so`

† Also add the library with BLACS routines corresponding to the MPI used.

The following table lists computational libraries for Intel® 64 architecture applications that use ScaLAPACK or Cluster FFTs.

Computational Libraries for the Intel® 64 or Intel® Many Integrated Core Architecture

Function domain	Static Linking	Dynamic Linking
ScaLAPACK, LP64 interface‡	`libmkl_scalapack_lp64.a` `libmkl_core.a`	`libmkl_scalapack_lp64.so` `libmkl_core.so`
ScaLAPACK, ILP64 interface‡	`libmkl_scalapack_ilp64.a` `libmkl_core.a`	`libmkl_scalapack_ilp64.so` `libmkl_core.so`
Cluster Fourier Transform Functions‡	`libmkl_cdft_core.a` `libmkl_core.a`	`libmkl_cdft_core.so` `libmkl_core.so`

‡ Also add the library with BLACS routines corresponding to the MPI used.

See Also
Linking with ScaLAPACK and Cluster FFTs
Using the Link-line Advisor
Using the ILP64 Interface vs. LP64 Interface

Linking with Compiler Run-time Libraries

Dynamically link `libiomp5`, the compatibility OpenMP* run-time library, even if you link other libraries statically.

Linking to the `libiomp5` statically can be problematic because the more complex your operating environment or application, the more likely redundant copies of the library are included. This may result in performance issues (oversubscription of threads) and even incorrect results.

To link `libiomp5` dynamically, be sure the `LD_LIBRARY_PATH` environment variable is defined correctly.

See Also
Scripts to Set Environment Variables
Layered Model Concept

Linking with System Libraries

To use the Intel MKL FFT, Trigonometric Transform, or Poisson, Laplace, and Helmholtz Solver routines, link also the math support system library by adding " `-lm` " to the link line.

On Linux OS, the `libiomp5` library relies on the native `pthread` library for multi-threading. Any time `libiomp5` is required, add `-lpthread` to your link line afterwards (the order of listing libraries is important).

NOTE To link with Intel MKL statically using a GNU or PGI compiler, link also the system library `libdl` by adding `-ldl` to your link line. The Intel compiler always passes `-ldl` to the linker.

See Also
Linking Examples

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 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 builder uses static Intel MKL interface, threading, and core libraries to build a custom shared object for the IA-32 architecture.
`libintel64`	The builder uses static Intel MKL interface, threading, and core libraries to build a custom shared object for the Intel® 64 architecture.
`soia32`	The builder uses the single dynamic library `libmkl_rt.so` to build a custom shared object for the IA-32 architecture.
`sointel64`	The builder uses the single dynamic library `libmkl_rt.so` to build a custom shared object for the Intel® 64 architecture.
`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 interfacefor the Intel 64architecture.The default value is `lp64`.
`threading = {parallel| sequential}`	Defines whether to use the Intel 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_example_list` (no extension).
`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® 64 architecture.

See Also
Using the Single Dynamic Library

Composing a List of Functions

To compose a list of functions for a minimal custom shared object needed for your application, you can use the following procedure:

1. Link your application with installed Intel MKL libraries to make sure the application builds.
2. Remove all Intel MKL libraries from the link line and start linking.

   Unresolved symbols indicate Intel MKL functions that your application uses.
3. Include these functions in the list.

Important Each time your application starts using more Intel MKL functions, update the list to include the new functions.

See Also
Specifying Function Names

Specifying Function Names

In the file with the list of functions for your custom shared object, 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_
```

For more examples, see domain-specific lists of functions in the *`<mkl directory>`*`/tools/builder` folder.

NOTE The lists of functions are provided in the *`<mkl directory>`*`/tools/builder` folder merely as examples. See Composing a List of Functions for how to compose lists of functions for your custom shared object.

TIP Names of Fortran-style routines (BLAS, LAPACK, etc.) can be both upper-case or lower-case, with or without the trailing underscore. For example, these names are equivalent:
BLAS: `dgemm`, `DGEMM`, `dgemm_`, `DGEMM_`
LAPACK: `dgetrf`, `DGETRF`, `dgetrf_`, `DGETRF_`.

Properly capitalize names of C support functions in the function list. To do this, follow the guidelines below:

1. In the `mkl_service.h` include file, look up a `#define` directive for your function (`mkl_service.h` is included in the `mkl.h` header file).
2. Take the function name from the replacement part of that directive.

For example, the `#define` directive for the `mkl_disable_fast_mm` function is `#define mkl_disable_fast_mm MKL_Disable_Fast_MM`.

Capitalize the name of this function in the list like this: `MKL_Disable_Fast_MM`.

For the names of the Fortran support functions, see the tip.

NOTE If selected functions have several processor-specific versions, the builder automatically includes them all in the custom library and the dispatcher manages them.

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

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Improving Performance with Threading

Intel 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 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: `?dttrsb, ?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® Core™2 Duo and Intel® 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® 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 complex-to-complex transforms using split-complex layout 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 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 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 `libiomp5`. 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.

Using the `mkl_set_num_threads` and `mkl_domain_set_num_threads` functions to control parallelism of Intel MKL from parallel user threads may result in a race condition that impacts the performance of the application because these functions operate on internal control variables that are global, that is, apply to all threads. For example, if parallel user threads call these functions to set different numbers of threads for the same function domain, the number of threads actually set is unpredictable. To avoid this kind of data races, use the `mkl_set_num_threads_local` function (see the "Support Functions" chapter in the *Intel MKL Reference Manual* for the function description).

See Also

Using Additional Threading Control
Linking with Compiler Run-time Libraries

Techniques to Set the Number of Threads

Use the following techniques to specify the number of threads to use in Intel 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()`
  - `mkl_set_num_threads_local()`

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 settings. The exception, which is a consequence of the previous rule, is that a call to the OpenMP subroutine `omp_set_num_threads()` does not have precedence over the settings of 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:
  ```
  setenv 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
int main(int args, char *argv[]){
double *a, *b, *c;
a = (double*)malloc(sizeof(double)*SIZE*SIZE);
b = (double*)malloc(sizeof(double)*SIZE*SIZE);
c = (double*)malloc(sizeof(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]);
}
free (a);
free (b);
free (c);
return 0;
}
```

```
// ******* Fortran language *******
PROGRAM DGEMM_DIFF_THREADS
```

```
INTEGER N, I, J
PARAMETER (N=100)
REAL*8 A(N,N),B(N,N),C(N,N)
REAL*8 ALPHA, BETA

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 MKL-specific Environment Variables for Threading Control

Intel MKL provides optional threading controls, that is, the environment variables and support functions that are independent of OpenMP. They behave similar to their OpenMP equivalents, but take precedence over them in the meaning that the Intel 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 of 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	Support Function	Comment	Equivalent OpenMP* Environment Variable
`MKL_NUM_THREADS`	`mkl_set_num_threads` `mkl_set_num_threads_local`	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 Intel 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 using the Intel® Hyper-Threading Technology), 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_DOMAIN_ALL | MKL_DOMAIN_BLAS | MKL_DOMAIN_FFT |
MKL_DOMAIN_VML | MKL_DOMAIN_PARDISO

<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, values of *`<MKL-domain-env-name>`* indicate function domains as follows:

`MKL_DOMAIN_ALL`	All function domains
`MKL_DOMAIN_BLAS`	BLAS Routines
`MKL_DOMAIN_FFT`	non-cluster Fourier Transform Functions
`MKL_DOMAIN_VML`	Vector Mathematical Functions
`MKL_DOMAIN_PARDISO`	PARDISO

For example,

```
MKL_DOMAIN_ALL 2 : MKL_DOMAIN_BLAS 1 : MKL_DOMAIN_FFT 4
```

```
MKL_DOMAIN_ALL=2 : MKL_DOMAIN_BLAS=1 : MKL_DOMAIN_FFT=4

MKL_DOMAIN_ALL=2,  MKL_DOMAIN_BLAS=1,  MKL_DOMAIN_FFT=4

MKL_DOMAIN_ALL=2;  MKL_DOMAIN_BLAS=1;  MKL_DOMAIN_FFT=4

MKL_DOMAIN_ALL  = 2  MKL_DOMAIN_BLAS 1 ,  MKL_DOMAIN_FFT  4

MKL_DOMAIN_ALL,2: MKL_DOMAIN_BLAS 1, MKL_DOMAIN_FFT,4 .
```

The global variables `MKL_DOMAIN_ALL`, `MKL_DOMAIN_BLAS`, `MKL_DOMAIN_FFT`, `MKL_DOMAIN_VML`, and `MKL_DOMAIN_PARDISO`, as well as the interface for the Intel 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_DOMAIN_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_DOMAIN_ALL= 1, MKL_DOMAIN_BLAS =4`	All parts of Intel MKL should try one thread, except for BLAS, which is suggested to try four threads.
`MKL_DOMAIN_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_DOMAIN_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_DOMAIN_BLAS );`" suggests the same, regardless of later calls to `mkl_set_num_threads()`. However, a function call with input "`MKL_DOMAIN_ALL`", such as "`mkl_domain_set_num_threads (4, MKL_DOMAIN_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_DOMAIN_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_DOMAIN_BLAS=4,MKL_DOMAIN_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_DOMAIN_BLAS );

mkl_domain_set_num_threads ( 2, MKL_DOMAIN_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 `setenv` commands, depending on the shell you use.

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_DOMAIN_ALL=1, MKL_DOMAIN_BLAS=4"

export MKL_DYNAMIC=FALSE
```

For the csh or tcsh shell, use the `setenv` commands:

`setenv <VARIABLE NAME> <value>`.

For example:

```
setenv MKL_NUM_THREADS 4
setenv MKL_DOMAIN_NUM_THREADS "MKL_DOMAIN_ALL=1, MKL_DOMAIN_BLAS=4"
setenv MKL_DYNAMIC FALSE
```

Other Tips and Techniques to Improve Performance

See Also

Improving Performance on Intel Xeon Phi Coprocessors Tips for Intel® Many Integrated Core Architecture

Coding Techniques

To improve performance of your application that calls Intel MKL, align your arrays on 64-byte boundaries and ensure that the leading dimensions of the arrays are divisible by 64.

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 $N*(N+1)/2$.

Hardware Configuration Tips

Dual-Core Intel® Xeon® processor 5100 series systems

To get the best performance with Intel MKL on Dual-Core Intel® Xeon® 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.

Intel® Hyper-Threading Technology

Intel® Hyper-Threading Technology (Intel® 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 Intel HT Technology.

If you run with Intel HT Technology 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

Improving Performance with Threading

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 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 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 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 Intel 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, 11, and 13.

Enabling Low-communication Algorithm in Cluster FFT

You may obtain a better performance of a one-dimensional Cluster FFT by enabling a low-communication Segment Of Interest FFT (SOI FFT) algorithm. To use this algorithm, you must set the `MKL_CFFT_ENABLE_SOI` environment variable to "1" or "yes".

CAUTION While using fewer MPI communications, the SOI FFT algorithm incurs a minor loss of precision (about one decimal digit).

Using Memory Management

Intel MKL Memory Management Software

Intel 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. To turn it off, set the `MKL_DISABLE_FAST_MM` environment variable to any value or call the `mkl_disable_fast_mm()` function. Be aware that this change may negatively impact performance of some Intel MKL routines, especially for small problem sizes.

Redefining Memory Functions

In C/C++ programs, you can replace Intel MKL memory functions that the library uses by default with your 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® Math Kernel Library (Intel® MKL) provides broad support for Fortran and C/C++ programming. However, not all functions support both Fortran and C interfaces. For example, some LAPACK functions have no C interface. You can call such functions from C using mixed-language programming.

If you want to use LAPACK or BLAS functions that support Fortran 77 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
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Using Language-Specific Interfaces with Intel® Math Kernel Library

This section discusses mixed-language programming and the use of language-specific interfaces with Intel MKL.

See also the "FFTW Interface to Intel® Math Kernel Library" Appendix 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 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.

File name	Contains
`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 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.
`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® 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 MKL provides the interface libraries and modules precompiled with the Intel® 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® 64 architecture,

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

Important The 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, the command

```
make libintel64 FC=pgf95 INSTALL_DIR=<userpgf95 dir> interface=lp64
```

builds the required library and `.mod` files and installs them in subdirectories of `<userpgf95 dir>`.

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

- For the IA-32 architecture,

  ```
  make cleania32 INSTALL_DIR=<user dir>
  ```
- For the Intel® 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 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 compiler, along with source code, to be used with different compilers.

Mixed-language Programming with the Intel Math Kernel Library

Appendix A: Intel(R) Math Kernel Library Language Interfaces Support lists the programming languages supported for each Intel 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 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.

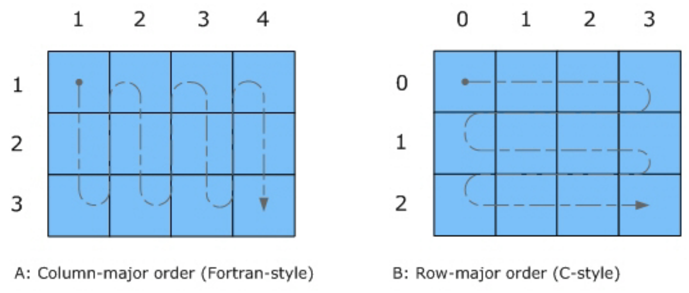
CAUTION Avoid calling BLAS 95/LAPACK 95 from C/C++. Such calls require skills in manipulating the descriptor of a deferred-shape array, which is the Fortran 90 type. Moreover, BLAS95/LAPACK95 routines contain links to a Fortran RTL.

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).


1
2
3
4
1
2
3
A: Column-major order (Fortran-style)
0
1
2
3
0
1
2
B: Row-major order (C-style)

For example, if a two-dimensional matrix A of size `mxn` 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-1)*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.h` header file with the C interface to LAPACK. `mkl.h` includes the `mkl_lapacke.h` header file, which 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 documentation for the Intel® 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 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>"
```

See Also

Intel® Software Documentation Library for the Intel® Fortran Compiler XE documentation

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 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
int main()
{
int n = N, inca = 1, incb = 1, i;
MKL_Complex16 a[N], b[N], c;
for( i = 0; i < n; i++ ){
a[i].real = (double)i; a[i].imag = (double)i * 2.0;
b[i].real = (double)(n - i); b[i].imag = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.real, c.imag );
return 0;
}
```

Example "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 <stdio.h>
#include "mkl.h"
typedef struct{ double re; double im; } complex16;
#define N 5
int 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 );
return 0;
}
```

Support for Boost uBLAS Matrix-matrix Multiplication

If you are used to uBLAS, you can perform BLAS matrix-matrix multiplication in C++ using Intel 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 and higher. 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 MKL Functions from Java* Applications

Intel MKL Java* Examples

To demonstrate binding with Java, Intel 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 Java* Examples

Running the Java* Examples

The Java examples support all the C and C++ compilers that Intel 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 thebash 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 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.

Obtaining Numerically Reproducible Results 7

Intel® Math Kernel Library (Intel® MKL) offers functions and environment variables that help you obtain Conditional Numerical Reproducibility (CNR) of floating-point results when calling the library functions from your application. These new controls enable Intel MKL to run in a special mode, when functions return bitwise reproducible floating-point results from run to run under the following conditions:

- Calls to Intel MKL occur in a single executable
- Input and output arrays in function calls are properly aligned
- The number of computational threads used by the library does not change in the run

It is well known that for general single and double precision IEEE floating-point numbers, the associative property does not always hold, meaning (a+b)+c may not equal a +(b+c). Let's consider a specific example. In infinite precision arithmetic $2^{-63} + 1 + -1 = 2^{-63}$. If this same computation is done on a computer using double precision floating-point numbers, a rounding error is introduced, and the order of operations becomes important:

$(2^{-63} + 1) + (-1) \simeq 1 + (-1) = 0$

versus

$2^{-63} + (1 + (-1)) \simeq 2^{-63} + 0 = 2^{-63}$

This inconsistency in results due to order of operations is precisely what the new functionality addresses.

The application related factors that affect the order of floating-point operations within a single executable program include selection of a code path based on run-time processor dispatching, alignment of data arrays, variation in number of threads, threaded algorithms and internal floating-point control settings. You can control most of these factors by controlling the number of threads and floating-point settings and by taking steps to align memory when it is allocated (see the Getting Reproducible Results with Intel® MKL knowledge base article for details). However, run-time dispatching and certain threaded algorithms did not allow users to make changes that can ensure the same order of operations from run to run.

Intel MKL does run-time processor dispatching in order to identify the appropriate internal code paths to traverse for the Intel MKL functions called by the application. The code paths chosen may differ across a wide range of Intel processors and Intel architecture compatible processors and may provide differing levels of performance. For example, an Intel MKL function running on an Intel® Pentium® 4 processor may run one code path, while on the latest Intel® Xeon® processor it will run another code path. This happens because each unique code path has been optimized to match the features available on the underlying processor. One key way that the new features of a processor are exposed to the programmer is through the instruction set architecture (ISA). Because of this, code branches in Intel MKL are designated by the latest ISA they use for optimizations: from the Intel® Streaming SIMD Extensions 2 (Intel® SSE2) to the Intel® Advanced Vector Extensions (Intel® AVX). The feature-based approach introduces a challenge: if any of the internal floating-point operations are done in a different order or are re-associated, the computed results may differ.

Dispatching optimized code paths based on the capabilities of the processor on which the code is running is central to the optimization approach used by Intel MKL. So it is natural that consistent results require some performance trade-offs. If limited to a particular code path, performance of Intel MKL can in some circumstances degrade by more than a half. To understand this, note that matrix-multiply performance nearly doubled with the introduction of new processors supporting Intel AVX instructions. Even if the code branch is not restricted, performance can degrade by 10-20% because the new functionality restricts algorithms to maintain the order of operations.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or

Getting Started with Conditional Numerical Reproducibility

Intel MKL offers functions and environment variables to increase your chances of getting reproducible results. You can configure Intel MKL using functions or environment variables, but the functions provide more flexibility.

The following specific examples introduce you to the conditional numerical reproducibility.

Intel CPUs supporting Intel AVX

To ensure Intel MKL calls return the same results on every Intel CPU supporting Intel AVX instructions:

1. Make sure that:
   - Your application uses a fixed number of threads
   - Input and output arrays in Intel MKL function calls are aligned properly
2. Do either of the following:
   - Call

     `mkl_cbwr_set(MKL_CBWR_AVX)`
   - Set the environment variable

     `MKL_CBWR_BRANCH = AVX`

NOTE On non-Intel CPUs and on Intel CPUs that do not support Intel AVX, this environment setting may cause results to differ because the `AUTO` branch is used instead, while the above function call returns an error and does not enable the CNR mode.

Intel CPUs supporting Intel SSE2

To ensure Intel MKL calls return the same results on every Intel CPU supporting Intel SSE2 instructions:

1. Make sure that:
   - Your application uses a fixed number of threads
   - Input and output arrays in Intel MKL function calls are aligned properly
2. Do either of the following:
   - Call

     `mkl_cbwr_set(MKL_CBWR_SSE2)`
   - Set the environment variable

     `MKL_CBWR_BRANCH = SSE2`

NOTE On non-Intel CPUs and on Intel CPUs that do not support Intel SSE2, this environment setting may cause results to differ because the `AUTO` branch is used instead, while the above function call returns an error and does not enable the CNR mode.

Intel or Intel compatible CPUs supporting Intel SSE2

On non-Intel CPUs, only the `MKL_CBWR_AUTO` and `MKL_CBWR_COMPATIBLE` options are supported for function calls and only `AUTO` and `COMPATIBLE` options for environment settings.

To ensure Intel MKL calls return the same results on all Intel or Intel compatible CPUs supporting Intel SSE2 instructions:

1. Make sure that:
   - Your application uses a fixed number of threads
   - Input and output arrays in Intel MKL function calls are aligned properly
2. Do either of the following:
   - Call

     ```
     mkl_cbwr_set(MKL_CBWR_COMPATIBLE)
     ```
   - Set the environment variable

     ```
     MKL_CBWR_BRANCH = COMPATIBLE
     ```

NOTE The special `MKL_CBWR_COMPATIBLE/COMPATIBLE` option is provided because Intel and Intel compatible CPUs have two approximation instructions(rcpps/rsqrtps) that may return different results. This option ensures that Intel MKL does not use these instructions and forces a single Intel SSE2 only code path to be executed.

Next steps

See Specifying the Code Branches	for details of specifying the branch using environment variables.

See the following sections in the *Intel MKL Reference Manual*:

Support Functions for Conditional Numerical Reproducibility	for how to configure the CNR mode of Intel MKL using functions.
PARDISO* - Parallel Direct Sparse Solver Interface	for how to configure the CNR mode for PARDISO.

See Also
Code Examples

Specifying the Code Branches

Intel MKL provides conditional numerically reproducible results for a code branch determined by the supported instruction set architecture (ISA). The values you can specify for the `MKL_CBWR` environment variable may have one of the following equivalent formats:

- `MKL_CBWR="`*`<branch>`*`"`
- `MKL_CBWR="BRANCH=`*`<branch>`*`"`

The *`<branch>`* placeholder specifies the CNR branch with one of the following values:

Value	Description
`AUTO`	CNR mode uses:

Value	Description
	• The standard ISA-based dispatching model on Intel processors while ensuring fixed cache sizes, deterministic reductions, and static scheduling • The branch corresponding to `COMPATIBLE` otherwise
	CNR mode uses the branch for the following ISA:
`COMPATIBLE`	Intel® Streaming SIMD Extensions 2 (Intel® SSE2) without rcpps/rsqrtps instructions
`SSE2`	Intel SSE2
`SSE3`	Intel® Streaming SIMD Extensions 3 (Intel® SSE3)
`SSSE3`	Supplemental Streaming SIMD Extensions 3 (SSSE3)
`SSE4_1`	Intel® Streaming SIMD Extensions 4-1 (Intel® SSE4-1)
`SSE4_2`	Intel® Streaming SIMD Extensions 4-2 (Intel® SSE4-2)
`AVX`	Intel® Advanced Vector Extensions (Intel® AVX)
`AVX2`	Intel® Advanced Vector Extensions 2 (Intel® AVX2)

When specifying the CNR branch, be aware of the following:

- Reproducible results are provided under certain conditions.
- Settings other than `AUTO` or `COMPATIBLE` are available only for Intel processors. The code branch specified by `COMPATIBLE` is run on all non-Intel processors.
- To get the CNR branch optimized for the processor where your program is currently running, choose the value of `AUTO` or call the `mkl_cbwr_get_auto_branch` function.

Setting the `MKL_CBWR` environment variable or a call to an equivalent `mkl_set_cbwr_branch` function fixes the code branch and sets the reproducibility mode.

- If the value of the branch is incorrect or your processor does not support the specified branch, CNR ignores this value and uses the `AUTO` branch without providing any warning messages.
- Calls to functions that define the behavior of CNR must precede any of the math library functions that they control.
- Settings specified by the functions take precedence over the settings specified by the environment variable.

See the *Intel MKL Reference Manual* for how to specify the branches using functions.

See Also
Getting Started with Conditional Numerical Reproducibility

Reproducibility Conditions

Reproducible results are provided under these conditions:

- The number of threads is fixed and constant.

  Specifically:

  - If you are running your program on different processors, explicitly specify the number of threads.
  - To ensure that your application has deterministic behavior with OpenMP* parallelization and does not adjust the number of threads dynamically at run time, set `MKL_DYNAMIC` and `OMP_DYNAMIC` to FALSE. This is especially needed if you are running your program on different systems.

- Input and output arrays are aligned on 128-byte boundaries.

  Instead of the general 128-byte alignment, you can use a more specific alignment depending on the ISA. For example: 16-byte alignment suffices for Intel SSE2 or higher and 32-byte alignment suffices for Intel AVX or Intel AVX2. To ensure proper alignment of arrays, allocate memory for them using `mkl_malloc`.

Setting the Environment Variable for Conditional Numerical Reproducibility

The following examples illustrate the use of the `MKL_CBWR` environment variable. The first command in each list sets Intel MKL to run in the CNR mode based on the default dispatching for your platform. The other two commands in each list are equivalent and set the CNR branch to Intel AVX.

For the bash shell:

- `export MKL_CBWR="AUTO"`
- `export MKL_CBWR="AVX"`
- `export MKL_CBWR="BRANCH=AVX"`

For the C shell (csh or tcsh):

- `setenv MKL_CBWR "AUTO"`
- `setenv MKL_CBWR "AVX"`
- `setenv MKL_CBWR "BRANCH=AVX"`

See Also

Specifying the Code Branches

Code Examples

The following simple programs show how to obtain reproducible results from run to run of Intel MKL functions. See the *Intel MKL Reference Manual* for more examples.

C Example of CNR

```
#include <mkl.h>
int main(void) {
    int my_cbwr_branch;
    /* Align all input/output data on 128-byte boundaries to get reproducible results of Intel MKL
function calls */
    void *darray;
    int darray_size=1000;
    /* Set alignment value in bytes */
    int alignment=128;
    /* Allocate aligned array */
    darray = mkl_malloc (sizeof(double)*darray_size, alignment);
    /* Find the available MKL_CBWR_BRANCH automatically */
    my_cbwr_branch = mkl_cbwr_get_auto_branch();
    /* User code without Intel MKL calls */
    /* Piece of the code where CNR of Intel MKL is needed */
    /* The performance of Intel MKL functions might be reduced for CNR mode */
    if (mkl_cbwr_set(my_cbwr_branch)) {
        printf("Error in setting MKL_CBWR_BRANCH! Aborting…\n");
        return;
    }
    /* CNR calls to Intel MKL + any other code with aligned input\output data */
    /* Free the allocated aligned array */
    mkl_free(darray);
}
```

Fortran Example of CNR

```
    PROGRAM MAIN
    INCLUDE 'mkl.fi'
    INTEGER*4 MY_CBWR_BRANCH
! Align all input/output data on 128-byte boundaries to get reproducible results of Intel MKL function
calls
! Declare Intel MKL memory allocation routine
#ifdef _IA32
    INTEGER MKL_MALLOC
#else
    INTEGER*8 MKL_MALLOC
#endif
    EXTERNAL MKL_MALLOC, MKL_FREE
    DOUBLE PRECISION DARRAY
    POINTER (P_DARRAY,DARRAY(1))
    INTEGER DARRAY_SIZE
    PARAMETER (DARRAY_SIZE=1000)
! Set alignment value in bytes
    INTEGER ALIGNMENT
    PARAMETER (ALIGNMENT=128)
! Allocate aligned array
    P_DARRAY = MKL_MALLOC (%VAL(8*DARRAY_SIZE), %VAL(ALIGNMENT));
! Find the available MKL_CBWR_BRANCH automatically
    MY_CBWR_BRANCH = MKL_CBWR_GET_AUTO_BRANCH()
! User code without Intel MKL calls
! Piece of the code where CNR of Intel MKL is needed
! The performance of Intel MKL functions may be reduced for CNR mode
    IF (MKL_CBWR_SET (MY_CBWR_BRANCH) .NE. MKL_CBWR_SUCCESS) THEN
        PRINT *, 'Error in setting MKL_CBWR_BRANCH! Aborting…'
        RETURN
    ENDIF
! CNR calls to Intel MKL + any other code
! Free the allocated aligned array
    CALL MKL_FREE(P_DARRAY)

    END
```

Coding Tips 8

This section provides coding tips for managing data alignment and version-specific compilation.

Example of Data Alignment

Needs for best performance with Intel MKL or for reproducible results from run to run of Intel MKL functions require alignment of data arrays. The following example shows how to align an array on 64-byte boundaries. To do this, use `mkl_malloc()` in place of system provided memory allocators, as shown in the code example below.

Aligning Addresses on 64-byte Boundaries

```
// ******* C language *******
...
#include <stdlib.h>
#include <mkl.h>
...
void *darray;
int workspace;
// Set value of alignment
int alignment=64;
...
// Allocate aligned workspace
darray = mkl_malloc( sizeof(double)*workspace, alignment );
...
// call the program using MKL
mkl_app( darray );
...
// Free workspace
mkl_free( darray );
```

```
! ******* Fortran language *******
...
! Set value of alignment
integer    alignment
parameter (alignment=64)
...
! Declare Intel MKL routines
#ifdef _IA32
integer mkl_malloc
#else
integer*8 mkl_malloc
#endif
external mkl_malloc, mkl_free, mkl_app
...
double precision darray
pointer (p_wrk,darray(1))
integer workspace
...
! Allocate aligned workspace
p_wrk = mkl_malloc( %val(8*workspace), %val(alignment) )
...
! call the program using Intel MKL
call mkl_app( darray )
...
! Free workspace
call mkl_free(p_wrk)
```

Using Predefined Preprocessor Symbols for Intel® MKL Version-Dependent Compilation

Preprocessor symbols (macros) substitute values in a program before it is compiled. The substitution is performed in the preprocessing phase.

The following preprocessor symbols are available:

Predefined Preprocessor Symbol	Description
`__INTEL_MKL__`	Intel MKL major version
`__INTEL_MKL_MINOR__`	Intel MKL minor version
`__INTEL_MKL_UPDATE__`	Intel MKL update number
`INTEL_MKL_VERSION`	Intel MKL full version in the following format: `INTEL_MKL_VERSION = (__INTEL_MKL__*100+__INTEL_MKL_MINOR__)*100+__INTEL_MKL_UPDATE__`

These symbols enable conditional compilation of code that uses new features introduced in a particular version of the library.

To perform conditional compilation:

1. Include in your code the file where the macros are defined:
   - `mkl.h` for C/C++
   - `mkl.fi` for Fortran
2. [Optionally] Use the following preprocessor directives to check whether the macro is defined:
   - `#ifdef, #endif` for C/C++
   - `!DEC$IF DEFINED, !DEC$ENDIF` for Fortran
3. Use preprocessor directives for conditional inclusion of code:
   - `#if, #endif` for C/C++
   - `!DEC$IF, !DEC$ENDIF` for Fortran

Example

This example shows how to compile a code segment conditionally for a specific version of Intel MKL. In this case, the version is 10.3 Update 4:

C/C++:

```
#include "mkl.h"
#ifdef INTEL_MKL_VERSION
#if INTEL_MKL_VERSION ==  100304
//     Code to be conditionally compiled
#endif
#endif
```

Fortran:

```
    include "mkl.fi"
!DEC$IF DEFINED INTEL_MKL_VERSION
!DEC$IF INTEL_MKL_VERSION .EQ. 100304
*     Code to be conditionally compiled
!DEC$ENDIF
!DEC$ENDIF
```

Working with the Intel® Math Kernel Library Cluster Software

9

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Linking with ScaLAPACK and Cluster FFTs

The Intel 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 Intel 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 and programming interface (LP64 or ILP64). Available libraries are listed in Directory Structure in Detail. For example, for the 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 used. 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; specifically, for Intel MPI, 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 linker script>	A linker script that corresponds to the MPI version.

For example, if you are using Intel MPI, want to statically link with ScaLAPACK using the LP64 interface, 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 Link-line Advisor to quickly choose the appropriate set of *<MKL cluster Library>*, *<BLACS>*, and *<MKL core libraries>*.

See Also

Linking Your Application with the Intel® 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 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 does not set the default number of threads to one, but depends on the OpenMP libraries used with the compiler to set the default number. For the threading layer based on the Intel 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 number of MPI ranks per node multiplied by the number of threads per node should not exceed the number of hardware threads per node.

If you are using your login environment to set an environment variable, such as `OMP_NUM_THREADS`, remember that changing the 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 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

Techniques to Set the Number of Threads

Using Shared Libraries

All needed shared libraries must be visible on all nodes at run time. To achieve this, set the `LD_LIBRARY_PATH` environment variable accordingly.

If Intel MKL is installed only on one node, link statically when building your Intel MKL applications rather than use shared libraries.

The Intel® 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® 64 or Intel® Many Integrated Core architecture, add `libmkl_scalapack_lp64.a` or `libmkl_scalapack_ilp64.a`, depending on 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 under the following conditions:

- Main module is in C.
- You are using the Intel® C++ Compiler.
- You are using MPICH2.
- The `PATH` environment variable contains a directory with the MPI linker scripts.
- `$MKLPATH` is a user-defined variable containing `<mkl_directory>/lib/ia32`.

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

```
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:

```
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
Using the Link-line Advisor

Examples for Linking a Fortran Application

These examples illustrate linking of an application under the following conditions:

- Main module is in Fortran.
- Intel® Fortran Compiler is used.
- Intel® MPI is used.
- The `PATH` environment variable contains a directory with the MPI linker scripts.
- `$MKLPATH` is a user-defined variable containing `<mkl directory>/lib/intel64` .

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

```
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® 64 architecture, use the following link line:

```
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
Using the Link-line Advisor

Using Intel® Math Kernel Library on Intel® Xeon Phi™ Coprocessors

10

Intel® Math Kernel Library (Intel® MKL) offers two sets of libraries to support Intel® Many Integrated Core (Intel® MIC) Architecture:

- For the host computer based on Intel® 64 or compatible architecture and running a Linux* operating system
- For Intel® Xeon Phi™ coprocessors

You can control how Intel MKL offloads computations to Intel® Xeon Phi™ coprocessors. Either you can offload computations automatically or use Compiler Assisted Offload:

- Automatic Offload.

  On Linux* OS running on Intel® 64 or compatible architecture systems, Automatic Offload automatically detects the presence of coprocessors based on Intel MIC Architecture and automatically offloads computations that may benefit from additional computational resources available. This usage model allows you to call Intel MKL routines as you would normally with minimal changes to your program. The only change needed to enable Automatic Offload is either the setting of an environment variable or a single function call. For details see Automatic Offload.
- Compiler Assisted Offload.

  This usage model enables you to use the Intel compiler and its offload pragma support to manage the functions and data offloaded to a coprocessor. Within an offload region, you should specify both the input and output data for the Intel MKL functions to be offloaded. After linking with the Intel MKL libraries for Intel MIC Architecture, the compiler provided run-time libraries transfer the functions along with their data to a coprocessor to carry out the computations. For details see Compiler Assisted Offload.

In addition to offloading computations to coprocessors, you can call Intel MKL functions from an application that runs natively on a coprocessor. Native execution occurs when an application runs entirely on Intel MIC Architecture. Native mode is a fast way to make an existing application run on Intel MIC Architecture with minimal changes to the source code. For more information, see Running Intel MKL on an Intel Xeon Phi Coprocessor in Native Mode.

Intel MKL functionality offers different levels of support for Intel MIC Architecture:

- **Optimized** - the following components are tuned for the Intel MIC Architecture:
  - Several BLAS (level 1, 2, and 3), Sparse BLAS, and LAPACK routines.
  - Vector Math Library (VML) and Vector Statistical Library (VSL), including random number generators (RNG).
  - Fast Fourier transforms.

  Please see the *Release Notes* for details.
- **Unsupported** - Poisson Library, Iterative Sparse Solvers, and Trust Region Solvers.
- **Supported** - all other functionality runs on Intel Xeon Phi coprocessors.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain

Optimization Notice
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Automatic Offload

Automatic Offload provides performance improvements with fewer changes to the code than Compiler Assisted Offload. If you are executing a function on the host CPU, Intel MKL running in the Automatic Offload mode may offload part of the computations to one or multiple Intel Xeon Phi coprocessors without you explicitly offloading computations. By default, Intel MKL determines the best division of the work between the host CPU and coprocessors. However, you can specify a custom work division.

To enable Automatic Offload and control the division of work, use environment variables or support functions. See the *Intel MKL Reference Manual* for detailed descriptions of the support functions.

Important Use of Automatic Offload does not require changes in your link line.

Automatic Offload Controls

The table below lists the environment variables for Automatic Offload and the functions that cause *similar* results. See the *Intel MKL Reference Manual* for detailed descriptions of the functions. To control the division of work between the host CPU and Intel Xeon Phi coprocessors, the environment variables use a fractional measure ranging from zero to one.

Environment Variable	Support Function	Description	Value
`MKL_MIC_ENABLE`	`mkl_mic_enable`	Enables Automatic Offload.	1
`OFFLOAD_DEVICES`	None	Specifies a list of coprocessors to be used for Automatic Offload. In particular, this setting may help you to configure the environment for an MPI application to run Intel MKL in the Automatic Offload mode. If this variable is not set, all the coprocessors available on the system are used for Automatic Offload. You can set this environment variable if Automatic Offload is enabled by the environment setting or function call.	A comma-separated list of integers, each ranging from 0 to the largest number of an Intel Xeon Phi coprocessor on the system, with the maximum of 31. Values out of this range are ignored. Moreover, if the list contains any non-integer data, the list is ignored completely as if the environment variable were not set at all. For example, if your system has 4 Intel Xeon Phi coprocessors and the value of the list is 1,3, Intel MKL uses only coprocessors 1 and 3 for Automatic Offload, and Intel MKL support functions and environment variables refer to these coprocessors as coprocessors 0 and 1.

Environment Variable	Support Function	Description	Value
		Setting this variable to an empty value is equivalent to completely disabling Automatic Offload regardless of the value of `MKL_MIC_ENABLE`. After setting this environment variable, Intel MKL support functions and environment variables refer to the specified coprocessors by their indexes in the list, starting with zero. `OFFLOAD_DEVICES` is a common setting for Intel MKL and Intel® Compilers. So, refer to the *Intel® Compiler User and Reference Guides* for more information.	
`MKL_HOST_WORKDIVISION`	`mkl_mic_set_workdivision`	Specifies the fraction of work for the host CPU to do.	A floating-point number ranging from 0.0 to 1.0. For example, the value could be 0.2 or 0.33. Intel MKL ignores negative values and treats values greater than 1 as 1.0.
`MKL_MIC_WORKDIVISION`	`mkl_mic_set_workdivision`	Specifies the fraction of work to do on all the Intel Xeon Phi coprocessors on the system.	See `MKL_HOST_WORKDIVISION`
`MKL_MIC_<number>_WORKDIVISION`	`mkl_mic_set_workdivision`	Specifies the fraction of work to do on a specific Intel Xeon Phi coprocessor. Here *<number>* is an integer ranging from 0 to the largest number of an Intel Xeon Phi coprocessor on the system, with the maximum of 31. For example, if the system has two Intel Xeon Phi coprocessors, *<number>* can be 0 or 1.	See `MKL_HOST_WORKDIVISION`
`MKL_MIC_MAX_MEMORY`	`mkl_mic_max_memory`	Specifies the maximum coprocessor memory reserved for Automatic Offload computations on all of the Intel Xeon Phi	Memory size in Kilobytes (`K`), megabytes (`M`), gigabytes (`G`), or terabytes (`T`). For example, `MKL_MIC_MAX_MEMORY = 4096M` limits the coprocessor memory

Environment Variable	Support Function	Description	Value
		coprocessors on the system. Each process that performs Automatic Offload computations uses additional coprocessor memory specified by the environment variable.	reserved for Automatic Offload computations to 4096 megabytes or 4 gigabytes. Setting `MKL_MIC_MAX_MEMORY = 4G` specifies the same amount of memory in gigabytes.
`MKL_MIC_<number>_MAX_MEMORY`	`mkl_mic_max_memory`	Specifies the maximum coprocessor memory reserved for Automatic Offload computations on a specific Intel Xeon Phi coprocessor on the system. Here *<number>* is an integer ranging from 0 to the largest number of an Intel Xeon Phi coprocessor on the system, with the maximum of 31. For example, if the system has two Intel Xeon Phi coprocessors, <number> can be 0 or 1.	Memory size in Kilobytes (`K`), megabytes (`M`), gigabytes (`G`), or terabytes (`T`). For example, `MKL_MIC_MAX_MEMORY = 4096M` limits the coprocessor memory reserved for Automatic Offload computations to 4096 megabytes or 4 gigabytes. Setting `MKL_MIC_MAX_MEMORY = 4G` specifies the same amount of memory in gigabytes.
`OFFLOAD_REPORT`	`mkl_mic_set_offload_report`	Specifies the profiling report level for Automatic Offload. `OFFLOAD_REPORT` is a common setting for Intel MKL and Intel® Compilers. So, refer to the *Intel® Compiler User and Reference Guides* for more information. **NOTE** The `mkl_mic_set_offload_report` function enables you to turn profile reporting on/off at run time but does not change the reporting level.	An integer ranging from 0 to 2: 0 - No reporting, default. 1 - The report includes: • The name of the function called in the Automatic Offload (AO) mode. • Effective work division. The value of -1 indicates that the hint, that is, the work division specified by the `mkl_mic_set_workdivison` function or the appropriate `MKL_*_WORKDIVISION` environment variable was ignored in this function call. • The time spent on the host CPU during the call. • The time spent on each available Intel Xeon Phi coprocessor during the call. 2 - In addition to the above information, the report includes: • The amounts of data transferred to and from each available coprocessor during the call.

Environment Variable	Support Function	Description	Value
`LD_LIBRARY_PATH`	None	Search path for host-side dynamic libraries.	Must contain the path to host-side Intel MIC Platform Software Stack libraries used by Intel MKL. The default path is `/opt/intel/mic/coi/host-linux-release/lib`.
`MIC_LD_LIBRARY_PATH`	None	Search path for coprocessor-side dynamic libraries.	Must contain: • The path to coprocessor-side Intel MIC Platform Software Stack libraries used by Intel MKL. The default path is `/opt/intel/mic/coi/device-linux-release/lib`. • The path to Intel MKL coprocessor-side libraries. The default path is `<mkl directory>/lib/mic`.

- Settings specified by the functions take precedence over the settings specified by the respective environment variables.
- Intel MKL interprets the values of `MKL_HOST_WORKDIVISION`, `MKL_MIC_WORKDIVISION`, and `MKL_MIC_<number>_WORKDIVISION` as guidance toward dividing work between coprocessors, but the library may choose a different work division if necessary.
- For LAPACK routines, setting the fraction of work to any value other than 0.0 enables the specified processor for Automatic Offload mode. However Intel MKL LAPACK does not use the value specified to divide the workload. For example, setting the fraction to 0.5 has the same effect as setting the fraction to 1.0.

See Also

Setting Environment Variables for Automatic Offload

Intel® Software Documentation Library for Intel® Compiler User and Reference Guides

Setting Environment Variables for Automatic Offload

Important To use Automatic Offload:

- If you completed the Setting Environment Variables step of the Getting Started process, `MKL_MIC_ENABLE` is the only environment variable that you need to set.
- Otherwise, you must also set the `LD_LIBRARY_PATH` and `MIC_LD_LIBRARY_PATH` environment variables.

To set the environment variables for Automatic Offload mode, described in Automatic Offload Controls, use the appropriate commands in your command shell:

- For the bash shell, set the appropriate environment variable(s) as follows:

  ```
  export MKL_MIC_ENABLE=1
  ```

  ```
  export OFFLOAD_DEVICES=<list>
  ```

  For example: `export OFFLOAD_DEVICES=1,3`

```
export MKL_HOST_WORKDIVISION=<value>
```

For example: `export MKL_HOST_WORKDIVISION=0.2`

```
export MKL_MIC_WORKDIVISION=<value>
```

```
export MKL_MIC_<number>_WORKDIVISION=<value>
```

For example: `export MKL_MIC_2_WORKDIVISION=0.33`

```
export MKL_MIC_MAX_MEMORY=<value>
```

```
export MKL_MIC_<number>_MAX_MEMORY=<value>
```

For example:

```
export MKL_MIC_0_MAX_MEMORY=2G
```

```
export OFFLOAD_REPORT=<level>
```

For example: `export OFFLOAD_REPORT=2`

```
export LD_LIBRARY_PATH="/opt/intel/mic/coi/host-linux-release/lib:$
{LD_LIBRARY_PATH}"
```

```
export MIC_LD_LIBRARY_PATH="/opt/intel/mic/coi/device-linux-release/lib:${MKLROOT}/
lib/mic:${MIC_LD_LIBRARY_PATH}"
```

- For the C shell (csh or tcsh), set the appropriate environment variable(s) as follows:

```
setenv MKL_MIC_ENABLE 1
```

```
setenv OFFLOAD_DEVICES <list>
```

For example: `setenv OFFLOAD_DEVICES 1,3`

```
setenv MKL_HOST_WORKDIVISION <value>
```

For example: `setenv MKL_HOST_WORKDIVISION 0.2`

```
setenv MKL_MIC_WORKDIVISION <value>
```

```
setenv MKL_MIC_<number>_WORKDIVISION <value>
```

For example: `setenv MKL_MIC_2_WORKDIVISION 0.33`

```
setenv MKL_MIC_MAX_MEMORY <value>
```

```
setenv MKL_MIC_<number>_MAX_MEMORY <value>
```

For example:

```
setenv MKL_MIC_0_MAX_MEMORY 2G
```

```
setenv OFFLOAD_REPORT <level>
```

For example: `setenv OFFLOAD_REPORT 2`

```
setenv LD_LIBRARY_PATH "/opt/intel/mic/coi/host-linux-release/lib:$
{LD_LIBRARY_PATH}"
```

```
setenv MIC_LD_LIBRARY_PATH "/opt/intel/mic/coi/device-linux-release/lib:${MKLROOT}/
lib/mic:${MIC_LD_LIBRARY_PATH}"
```

See Also

Automatic Offload Controls
Detailed Directory Structure of the lib/mic Directory

Compiler Assisted Offload

Compiler Assisted Offload is a method to offload computations to Intel Xeon Phi coprocessors that uses the Intel® compiler and its offload pragma support to manage the functions and data offloaded. See *Intel® Compiler User and Reference Guides* for more details.

Important The Intel compilers support Intel MIC Architecture starting with version 13.

See Also

Automatic Offload

Examples of Compiler Assisted Offload

The following are examples of Compiler Assisted Offload. Please see *Intel® Compiler User and Reference Guide* for more details.

These examples show how to call Intel MKL from offload regions that are executed on coprocessors based on Intel MIC Architecture and how to reuse data that already exists in the memory of the coprocessor and thus minimize data transfer.

Fortran

```
c      Upload A and B to the card, and do not deallocate them after the
c      pragma. C is uploaded and downloaded back, but the allocated memory
c      is retained
       !DEC$ ATTRIBUTES OFFLOAD : MIC :: SGEMM
       !DEC$ OFFLOAD TARGET( MIC:0 ) IN( N ), &
       !DEC$ IN( A: LENGTH( N * N ) ALLOC_IF(.TRUE.) FREE_IF(.FALSE.)), &
       !DEC$ IN( B: LENGTH( N * N ) ALLOC_IF(.TRUE.) FREE_IF(.FALSE.)), &
       !DEC$ INOUT( C: LENGTH( N * N ) ALLOC_IF(.TRUE.) FREE_IF(.FALSE.))
       CALL SGEMM( 'N', 'N', N, N, N, 1.0, A, N, B, N, 1.0, C, N )

c      Change C here

c      Reuse A and B on the card, and upload the new C. Free all the
c      memory on the card
       !DEC$ ATTRIBUTES OFFLOAD : MIC :: SGEMM
       !DEC$ OFFLOAD TARGET( MIC:0 ) IN( N ), &
       !DEC$ NOCOPY( A: LENGTH( N * N ) ALLOC_IF(.FALSE.) FREE_IF(.TRUE.)), &
       !DEC$ NOCOPY( B: LENGTH( N * N ) ALLOC_IF(.FALSE.) FREE_IF(.TRUE.)), &
       !DEC$ INOUT( C: LENGTH( N * N ) ALLOC_IF(.FALSE.) FREE_IF(.TRUE.))
       CALL SGEMM( 'N', 'N', N, N, N, 1.0, A, N, B, N, -1.0, C, N )
```

C

```
    /* Upload A and B to the card, and do not deallocate them after the pragma.
     * C is uploaded and downloaded back, but the allocated memory is retained. */
#pragma offload target(mic:0) \
    in(A: length(matrix_elements) alloc_if(1) free_if(0)) \
    in(B: length(matrix_elements) alloc_if(1) free_if(0)) \
    in(transa, transb, N, alpha, beta) \
    inout(C:length(matrix_elements) alloc_if(1) free_if(0))
    {
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
                &beta, C, &N);
    }

    /* Change C here */

    /* Reuse A and B on the card, and upload the new C. Free all the memory on
     * the card. */
#pragma offload target(mic:0) \
    nocopy(A: length(matrix_elements) alloc_if(0) free_if(1)) \
    nocopy(B: length(matrix_elements) alloc_if(0) free_if(1)) \
    in(transa, transb, N, alpha, beta) \
    inout(C:length(matrix_elements) alloc_if(0) free_if(1))
    {
        sgemm(&transa, &transb, &N, &N, &N, &alpha, A, &N, B, &N,
                &beta, C, &N);
    }
```

See Also

Intel® Software Documentation Library for Intel® Compiler User and Reference Guides

Linking on Intel Xeon Phi Coprocessors

Intel MKL provides both static and dynamic libraries for coprocessors based on Intel MIC Architecture, but the Single Dynamic Library is unavailable for the coprocessors.

See Selecting Libraries to Link with for libraries to list on your link line in the simplest case.

See Detailed Directory Structure of the lib/mic Directory for a full list of libraries provided in the `<mkl directory>/lib/mic` directory.

You can link either static or dynamic host libraries and either static or dynamic coprocessor-side libraries independently.

To run applications linked dynamically with the host-computer and coprocessor-side libraries, perform the Setting Environment Variables step of the Getting Started process, which sets:

- `LD_LIBRARY_PATH` to contain `<mkl directory>/lib/intel64`
- `MIC_LD_LIBRARY_PATH` to contain `<mkl directory>/lib/mic`

To make Intel MKL functions available on the coprocessor side, provide the `-offload-attribute-target=mic` option on your link line.

Important Because Intel MKL provides both LP64 and ILP64 interfaces, ensure that the host and coprocessor-side executables use the same interface or cast all 64-bit integers to 32-bit integers (or vice-versa) before calling coprocessor-side functions in your application.

The following examples illustrate linking with Intel compilers on Intel Xeon Phi coprocessors.

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

In these examples, `MKLINCLUDE` is defined as `$MKLROOT/include`
(if you successfully completed the Setting Environment Variables step of the Getting Started process, you can omit the `-I$MKLINCLUDE` parameter):

- Static linking of `myprog.f`, host-computer and coprocessor-side libraries for parallel Intel MKL using LP64 interface:

```
ifort myprog.f -I$MKLINCLUDE -offload-attribute-target=mic
-Wl,--start-group $MKLROOT/lib/intel64/libmkl_intel_lp64.a
$MKLROOT/lib/intel64/libmkl_intel_thread.a
$MKLROOT/lib/intel64/libmkl_core.a -Wl,--end-group
-openmp -lpthread -lm
-offload-option,mic,compiler,"-Wl,--start-group $MKLROOT/lib/mic/libmkl_intel_lp64.a
$MKLROOT/lib/mic/libmkl_intel_thread.a $MKLROOT/lib/mic/libmkl_core.a
-Wl,--end-group"
```

- Dynamic linking of `myprog.f`, host-computer and coprocessor-side libraries for parallel Intel MKL using LP64 interface:

```
ifort myprog.f -I$MKLROOT/include -offload-attribute-target=mic
-L$MKLROOT/lib/intel64
-lmkl_intel_lp64 -lmkl_intel_thread
-lmkl_core -openmp -lpthread -lm
-offload-option,mic,compiler,"-L$MKLROOT/lib/mic -lmkl_intel_lp64 -lmkl_intel_thread
-lmkl_core"
```

- Static linking of `myprog.f`, host-computer and coprocessor-side libraries for parallel Intel MKL using ILP64 interface:

```
ifort myprog.f -I$MKLINCLUDE -offload-attribute-target=mic
-Wl,--start-group $MKLROOT/lib/intel64/libmkl_intel_ilp64.a
```

```
$MKLROOT/lib/intel64/libmkl_intel_thread.a
$MKLROOT/lib/intel64/libmkl_core.a -Wl,--end-group
-openmp -lpthread -lm
-offload-option,mic,compiler,"-Wl,--start-group
$MKLROOT/lib/mic/libmkl_intel_ilp64.a $MKLROOT/lib/mic/libmkl_intel_thread.a
$MKLROOT/lib/mic/libmkl_core.a -Wl,--end-group"
```

- Dynamic linking of `myprog.f`, host-computer and coprocessor-side libraries for parallel Intel MKL using ILP64 interface:

```
ifort myprog.f -I$MKLROOT/include -offload-attribute-target=mic
-L$MKLROOT/lib/intel64
-lmkl_intel_ilp64 -lmkl_intel_thread
-lmkl_core -openmp -lpthread -lm
-offload-option,mic,compiler,"-L$MKLROOT/lib/mic -lmkl_intel_ilp64
-lmkl_intel_thread -lmkl_core"
```

- Static linking of `myprog.f`, host-computer and coprocessor-side libraries for sequential version of Intel MKL using LP64 interface:

```
ifort myprog.f -I$MKLINCLUDE -offload-attribute-target=mic
-Wl,--start-group $MKLROOT/lib/intel64/libmkl_intel_lp64.a
$MKLROOT/lib/intel64/libmkl_sequential.a
$MKLROOT/lib/intel64/libmkl_core.a -Wl,--end-group -lm
-offload-option,mic,compiler,"-Wl,--start-group $MKLROOT/lib/mic/libmkl_intel_lp64.a
$MKLROOT/lib/mic/libmkl_sequential.a $MKLROOT/lib/mic/libmkl_core.a -Wl,--end-group"
```

- Dynamic linking of `myprog.f`, host-computer and coprocessor-side libraries for sequential version of Intel MKL using LP64 interface:

```
ifort myprog.f -I$MKLROOT/include -offload-attribute-target=mic
-L$MKLROOT/lib/intel64
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lm
-offload-option,mic,compiler,"-L$MKLROOT/lib/mic -lmkl_intel_lp64 -lmkl_sequential
-lmkl_core"
```

See Also

Linking Your Application with the Intel® Math Kernel Library
Linking with System Libraries
Using the Link-line Advisor

Using Automatic Offload and Compiler Assisted Offload in One Application

You can use Automatic Offload and Compiler Assisted Offload in the same application. However, to avoid oversubscription of computational resources of the coprocessors, you should synchronize #pragma offload regions and calls of Intel MKL functions that support Automatic Offload.

Running Intel MKL on an Intel Xeon Phi Coprocessor in Native Mode

Concept of a Native Run

Some applications can benefit from running on Intel Xeon Phi coprocessors in native mode. In this mode, the application runs directly on a coprocessor and its Linux* operating system without being offloaded from a host system. To run on Intel MIC Architecture in the native mode, an application requires minimal changes to the source code.

Because in the native mode the code runs exclusively on a coprocessor, binaries built for native runs contain only the code to be run on a coprocessor. A specialized compiler option supports builds of applications to be run in the native mode.

To run an application that calls Intel MKL in the native mode, you need to perform these high-level steps:

1. On the host system, compile and build the application using the `-mmic` option.
2. Transfer the executable and all the dynamic libraries it requires to the coprocessor:
   - The Intel MKL libraries are available in the `<mkl directory>/lib/mic` directory.
   - `libiomp5.so` is available in the `<parent product directory>/compiler/lib/mic` directory.
3. Use the Secure Shell (SSH) or Telnet protocol to execute on the coprocessor and add the paths to the dynamic libraries transferred to the coprocessor in step 2 and to the value of the `LD_LIBRARY_PATH` environment variable.
4. Set OpenMP* thread affinity and the `OMP_NUM_THREADS` environment variable.
5. Execute just as you would on a standard Linux* system.

For more information, see *Intel® Compiler User and Reference Guides*.

See Also

Detailed Directory Structure of the lib/mic Directory
Improving Performance on Intel Xeon Phi Coprocessors
Notational Conventions

Linking with ScaLAPACK and Cluster FFTs on Intel Xeon Phi Coprocessors

The Intel MKL ScaLAPACK and Cluster FFTs support only Intel® MPI implementations for Intel MIC Architecture.

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 compiler wrappers to do this. For example, `mpiicc` or `mpiifort` are C or FORTRAN compiler wrappers, respectively, that use the correct header files and libraries. The locations of these wrappers and the MPI libraries depend on the Intel MPI version. For example, for the default installation of Intel MPI 4.1, the wrappers for the Intel MIC Architecture are `/opt/impi/mic/bin/mpiicc` and `/opt/impi/mic/bin/mpiifort`.

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

To link with Intel MKL ScaLAPACK and/or Cluster FFTs on Intel Xeon Phi coprocessors, use the link line 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.
- Enclose the cluster components, interface, threading, and computational libraries in grouping symbols `-Wl,--start-group` and `-Wl,--end-group`.

```
/opt/impi/mic/bin/mpi{icc|ifort} <files to link> -L<MKL path> -I<MKL include>

[-lmkl_scalapack_{lp64|ilp64}] [-lmkl_cdft_core]

-lmkl_blacs_intelmpi_{lp64|ilp64}

-lmkl_intel_{lp64|ilp64}

-lmkl_{intel_thread|sequential}

-lmkl_core

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

For example, if you are using Intel MPI 4.1, want to statically link with ScaLAPACK using the LP64 interface, and have only one MPI process per core (and thus do not use threading), specify the following linker options:

```
-L$MKLPATH -I$MKLINCLUDE $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
```

TIP Use the Link-line Advisor to quickly choose the appropriate set of libraries and linker options.

See Also
Linking Your Application with the Intel® Math Kernel Library
Setting the Number of Threads
Using Shared Libraries
Building ScaLAPACK Tests

Examples of Linking a C Application for Intel® Many Integrated Core Architecture

These examples illustrate linking of an application for Intel MIC Architecture under the following conditions:

- Main module is in C.
- *<path to mpi binaries>* is the path to Intel MPI binaries for Intel MIC Architecture.
- `$MKLPATH` is a user-defined variable containing *<mkl_directory>*`/lib/mic`.
- You are using the Intel® C++ Compiler.
- Your programming interface is LP64.

See the *Intel MKL Release Notes* for details of system requirements.

To link with ScaLAPACK for native runs on a cluster of systems based on the Intel MIC architecture, use the following link line:

```
<path to mpi binaries>/mpicc <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 native runs on a cluster of systems based on the Intel MIC architecture, use the following link line:

```
<path to mpi binaries>/mpicc <files to link>       \
   -Wl,--start-group                               \
   $MKLPATH/libmkl_cdft_core.a                     \
   $MKLPATH/libmkl_blacs_intelmpi_lp64.a           \
   $MKLPATH/libmkl_intel_lp64.a                    \
   $MKLPATH/libmkl_intel_thread.a                  \
   $MKLPATH/libmkl_core.a                          \
   -Wl,--end-group                                 \
   -liomp5 -lpthread
```

See Also
Working with the Intel® Math Kernel Library Cluster Software
Using the Link-line Advisor

Examples of Linking a Fortran Application for Intel® Many Integrated Core Architecture

These examples illustrate linking of an application for Intel MIC Architecture under the following conditions:

- Main module is in Fortran.
- *<path to mpi binaries>* is the path to Intel MPI binaries for Intel MIC Architecture.

- `$MKLPATH` is a user-defined variable containing *<mkl directory>*`/lib/mic`.
- You are using the Intel® Fortran Compiler.
- Your programming interface is LP64.

See the *Intel MKL Release Notes* for details of system requirements.

To link with ScaLAPACK for native runs on a cluster of systems based on the Intel MIC architecture, use the following link line:

```
<path to mpi binaries>/mpiifort <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 native runs on a cluster of systems based on the Intel MIC architecture, use the following link line:

```
<path to mpi binaries>/mpiifort <files to link>       \
   -Wl,--start-group                                  \
   $MKLPATH/libmkl_cdft_core.a                        \
   $MKLPATH/libmkl_blacs_intelmpi_lp64.a              \
   $MKLPATH/libmkl_intel_lp64.a                       \
   $MKLPATH/libmkl_intel_thread.a                     \
   $MKLPATH/libmkl_core.a                             \
   -Wl,--end-group                                    \
   -liomp5 -lpthread
```

See Also

Working with the Intel® Math Kernel Library Cluster Software
Using the Link-line Advisor

Threading Behavior of Intel MKL on Intel MIC Architecture

To avoid performance drops caused by oversubscribing Intel Xeon Phi coprocessors, Intel MKL limits the number of threads it uses to parallelize computations:

- For native runs on coprocessors, Intel MKL uses 4**Number-of-Cores* threads by default and scales down the number of threads back to this value if you request more threads and `MKL_DYNAMIC` is true.
- For runs that offload computations, Intel MKL uses 4*(*Number-of-Cores*-1) threads by default and scales down the number of threads back to this value if you request more threads and `MKL_DYNAMIC` is true.
- If you request fewer threads than the default number, Intel MKL will use the requested number.

Here *Number-of-Cores* is the number of Intel Xeon Phi coprocessors on the system.

See Also

MKL_DYNAMIC
Automatic Offload Controls
Techniques to Set the Number of Threads

Improving Performance on Intel Xeon Phi Coprocessors

To improve performance of Intel MKL on Intel Xeon Phi coprocessors, use the following tips, which are specific to Intel MIC Architecture. You can also use general performance improvement recommendations, in Coding Techniques.

Memory Allocation

Performance of many Intel MKL routines improves when input and output data reside in memory allocated with 2M pages because this enables you to address more memory with less pages and thus reduce the overhead of translating between virtual and physical memory addresses compared to memory allocated with the default page size of 4K. For more information, refer to *Intel® 64 and IA-32 Architectures Optimization Reference Manual* and *Intel® 64 and IA-32 Architectures Software Developer's Manual* (connect to http://www.intel.com/ and enter the name of each document in the **Find Content** text box).

To allocate memory with 2M pages, you can use the `mmap` system call with the `MAP_HUGETLB` flag.

To enable allocation of memory with 2M pages for data of size exceeding 2M and transferred with offload pragmas, set the `MIC_USE_2MB_BUFFERS` environment variable to 2M. See *Intel® Compiler User and Reference Guides* for more details.

Specifying the maximum coprocessor memory that can be used for Automatic Offload computations typically enhances the performance because Intel MKL can reserve and keep the memory on the coprocessor during Automatic Offload computations. You can specify the maximum memory by setting the `MKL_MIC_MAX_MEMORY` environment variable to a value such as 2 GB.

OpenMP and Threading Settings

To improve performance of Intel MKL routines, use the following OpenMP and threading settings:

- For BLAS, LAPACK, and Sparse BLAS:

  Set `KMP_AFFINITY=compact,granularity=fine`
- For FFT:

  Set `KMP_AFFINITY=balanced,granularity=fine` and use a number of threads that is a power of two

Data Alignment and Leading Dimensions

To improve performance of Intel MKL FFT functions, follow these recommendations:

- Align the first element of the input data on 64-byte boundaries
- For two- or higher-dimensional single-precision transforms, use leading dimensions (strides) divisible by 8 but not divisible by 16
- For two- or higher-dimensional double-precision transforms, use leading dimensions divisible by 4 but not divisible by 8

For other Intel MKL function domains, use general recommendations for data alignment.

See Also

Examples of Compiler Assisted Offload

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

11

Configuring the Eclipse* IDE CDT to Link with Intel MKL

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

TIP After configuring your CDT, you can benefit from the Eclipse-provided *code assist* feature. See Code/Context Assist description in the CDT Help for details.

To configure your Eclipse IDE CDT to link with Intel MKL, you need to perform the steps explained below. The specific instructions for performing these steps depend on your version of the CDT and on the tool-chain/compiler integration. Refer to the CDT Help for more details.

To configure your Eclipse IDE CDT, do the following:

1. Open **Project Properties** for your project.
2. Add the Intel MKL include path, that is, `<mkl directory>/include`, to the project's include paths.
3. Add the Intel MKL library path for the target architecture to the project's library paths. For example, for the Intel® 64 architecture, add `<mkl directory>/lib/intel64`.
4. Specify the names of the Intel MKL libraries to link with your application. For example, you may need the following libraries: `mkl_intel_lp64`, `mkl_intel_thread`, `mkl_core`, and `iomp5`.

NOTE Because compilers typically require library names rather than file names, omit the "`lib`" prefix and "`a`" or "`so`" extension.

See Also

Selecting Libraries to Link with
Linking in Detail

Getting Assistance for Programming in the Eclipse* IDE

Intel MKL provides an Eclipse* IDE plug-in (`com.intel.mkl.help`) that contains the Intel 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

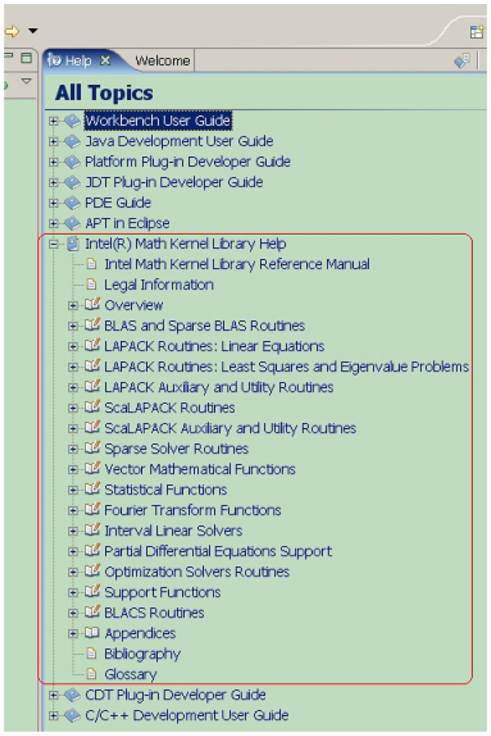
The Intel MKL plug-in for Eclipse IDE provides the first two features.

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

Viewing the Intel® Math Kernel Library Reference Manual in the Eclipse* IDE

To view the Reference Manual, in Eclipse,

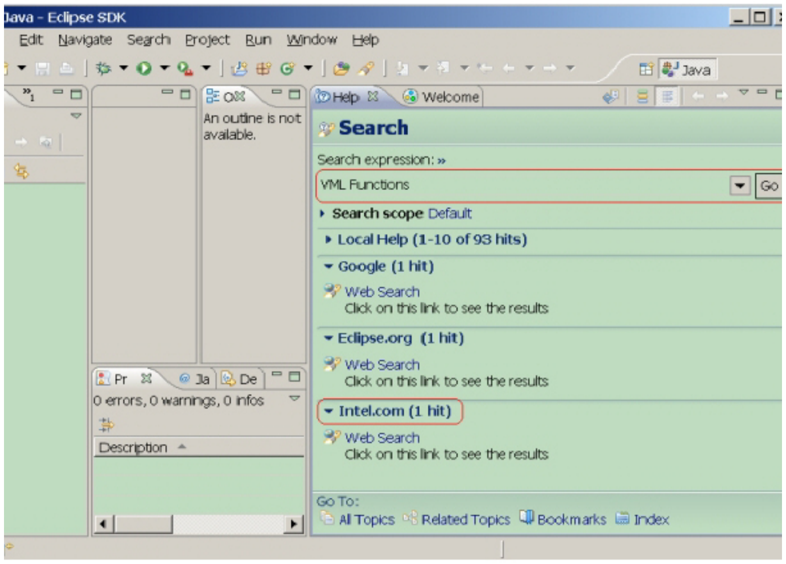
1. Select **Help** > **Help Contents** from the menu.
2. In the Help tab, under **All Topics** , click **Intel® Math Kernel Library Help** .
3. In the Help tree that expands, click **Intel Math Kernel Library Reference Manual**.
4. The Intel MKL Help Index is also available in Eclipse, and the Reference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



Workbench User Guide
Java Development User Guide
Platform Plug-in Developer Guide
JDT Plug-in Developer Guide
PDE Guide
APT in Eclipse
Intel(R) Math Kernel Library Help
Intel Math Kernel Library Reference Manual
Legal Information
Overview
BLAS and Sparse BLAS Routines
LAPACK Routines: Linear Equations
LAPACK Routines: Least Squares and Eigenvalue Problems
LAPACK Auxiliary and Utility Routines
ScaLAPACK Routines
ScaLAPACK Auxiliary and Utility Routines
Sparse Solver Routines
Vector Mathematical Functions
Statistical Functions
Fourier Transform Functions
Interval Linear Solvers
Partial Differential Equations Support
Optimization Solvers Routines
Support Functions
BLACS Routines
Appendices
Bibliography
Glossary
CDT Plug-in Developer Guide
C/C++ Development User Guide

Searching the Intel Web Site from the Eclipse* IDE

The Intel MKL plug-in tunes Eclipse Help search to targethttp://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.
Pr Ja De
0 errors, 0 warnings, 0 infos
Description
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
Go To:
All Topics Related Topics Bookmarks Index

LINPACK and MP LINPACK Benchmarks 12

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Intel® Optimized LINPACK Benchmark for Linux* OS

Intel® 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 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 on other Intel® software performance products, is available at http://www.intel.com/software/products/.

Contents of the Intel® Optimized LINPACK Benchmark

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

File in `./benchmarks/linpack/`	**Description**
`xlinpack_xeon32`	The 32-bit program executable for a system based on Intel® Xeon® processor or Intel® Xeon® processor MP with or without Streaming SIMD Extensions 3 (SSE3).
`xlinpack_xeon64`	The 64-bit program executable for a system with Intel Xeon processor using Intel® 64 architecture. This program may accelerate using Intel® Xeon Phi™ coprocessors if they are available on the system.
`xlinpack_mic`	The 64-bit program executable for a native run on an Intel Xeon Phi coprocessor.

File in `./benchmarks/linpack/`	**Description**
`runme_xeon32`	A sample shell script for executing a pre-determined problem set for `xlinpack_xeon32`.
`runme_xeon64`	A sample shell script for executing a pre-determined problem set for `xlinpack_xeon64`.
`runme_xeon64_ao`	A sample shell script for executing a pre-determined problem set for `xlinpack_xeon64`. The script enables acceleration by offloading computations to Intel Xeon Phi coprocessors available on the system.
`runme_mic`	A sample shell script for executing a pre-determined problem set for `xlinpack_mic`.
`lininput_xeon32`	Input file for a pre-determined problem for the `runme_xeon32` script.
`lininput_xeon64`	Input file for a pre-determined problem for the `runme_xeon64` script.
`lininput_xeon64_ao`	Input file for a pre-determined problem for the `runme_xeon64_ao` script.
`lininput_mic`	Input file for a pre-determined problem for the `runme_mic` script.
`lin_xeon32.txt`	Result of the `runme_xeon32` script execution.
`lin_xeon64.txt`	Result of the `runme_xeon64` script execution.
`lin_xeon64_ao.txt`	Result of the `runme_xeon64_ao` script execution.
`lin_mic.txt`	Result of the `runme_mic` 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
./runme_mic
```

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
./xlinpack_mic -e
```

The pre-defined data input files `lininput_xeon32`, `lininput_xeon64`, and `lininput_mic` are provided merely as examples. Different systems have different numbers of processors or amounts 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
`lininput_mic`	8 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 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 yet match the situation for your machine, edit the script.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Known Limitations of the Intel® 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 the Intel® 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® Optimized MP LINPACK Benchmark for Clusters

Overview of the Intel® Optimized MP LINPACK Benchmark for Clusters

The Intel® 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® 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® 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 processors more easily than with the HPL benchmark. Use the Intel Optimized MP LINPACK Benchmark to benchmark your cluster. The prebuilt binaries require Intel® MPI 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 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® 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.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Contents of the Intel® 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 of the Intel 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).

Directory/File in `./benchmarks/mp_linpack/`	Contents
`src/blas/HPL_dgemm.c`	HPL 2.0 code modified to capture `DGEMM` information, if desired, from `ASYOUGO2_DISPLAY`.
`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® 64 architecture and Linux OS.
`HPL.dat`	A repeat of `testing/ptest/HPL.dat` in the top-level directory.
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® MPI.
`bin_intel/ia32/xhpl_ia32_dynamic`	(New) Prebuilt binary for the IA-32 architecture and Linux OS. Dynamically linked against Intel® MPI.
`bin_intel/intel64/xhpl_intel64`	(New) Prebuilt binary for the Intel® 64 architecture and Linux OS. Statically linked against Intel® MPI.
`bin_intel/intel64/xhpl_intel64_dynamic`	(New) Prebuilt binary for the Intel® 64 architecture and Linux OS. Dynamically linked against Intel® MPI.
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® MPI.
`bin_intel/ia32/xhpl_hybrid_ia32_dynamic`	(New) Prebuilt hybrid binary for the IA-32 architecture and Linux OS. Dynamically linked against Intel® MPI.
`bin_intel/intel64/xhpl_hybrid_intel64`	(New) Prebuilt hybrid binary for the Intel® 64 architecture and Linux OS. Statically linked against Intel® MPI.
`bin_intel/intel64/xhpl_hybrid_intel64_dynamic`	(New) Prebuilt hybrid binary for the Intel® 64 and Linux OS. Dynamically linked against Intel® MPI.
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.
`lib_hybrid/intel64/libhpl_hybrid.a`	(New) Prebuilt library with the hybrid version of MP LINPACK for the Intel® 64 architecture and Intel MPI.
Files that refer to run scripts	

Directory/File in `./benchmarks/mp_linpack/`	**Contents**
`bin_intel/ia32/runme_ia32`	(New) Sample run script for the IA-32 architecture and a pure MPI binary statically linked against Intel MPI.
`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.
`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.
`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.
`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® 64 architecture and a pure MPI binary statically linked against Intel MPI.
`bin_intel/intel64/runme_intel64_dynamic`	(New) Sample run script for the Intel® 64 architecture and a pure MPI binary dynamically linked against Intel MPI.
`bin_intel/intel64/HPL_serial.dat`	(New) Example of an MP LINPACK benchmark input file for a pure MPI binary and the Intel® 64 architecture.
`bin_intel/intel64/runme_hybrid_intel64`	(New) Sample run script for the Intel® 64 architecture and a hybrid binary statically linked against Intel MPI.
`bin_intel/intel64/runme_hybrid_intel64_dynamic`	(New) Sample run script for the Intel® 64 architecture and a hybrid binary dynamically linked against Intel MPI.
`bin_intel/intel64/HPL_hybrid.dat`	(New) Example of an MP LINPACK benchmark input file for a hybrid binary and the Intel® 64 architecture.
`nodeperf.c`	(New) Sample utility that tests the `DGEMM` speed across the cluster.

‡ For a list of supported versions of Intel MPI, see system requirements in Release Notes.

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 Intel 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® 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® 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 MKL and prebuilt MP LINPACK hybrid libraries.

CAUTION Use this option only with an Intel® compiler and the Intel® MPI library version 3.1 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 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
Notational Conventions

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® Math Kernel Library Language Interfaces Support

A

Language Interfaces Support, by Function Domain

The following table shows language interfaces that Intel® Math Kernel Library (Intel® 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® 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
Trigonometric Transform routines		Yes	Yes
Fast Poisson, Laplace, and Helmholtz Solver (Poisson Library) routines		Yes	Yes
Optimization (Trust-Region) Solver routines	Yes	Yes	Yes
Data Fitting functions	Yes	Yes	Yes
Support functions (including memory allocation)	Yes	Yes	Yes

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

Include Files

The table below lists Intel MKL include files.

NOTE The `*.f90` include files supersede the `*.f77` include files and can be used for FORTRAN 77 as well as for later versions of Fortran. However, the `*.f77` files are kept for backward compatibility.

Function domain	Fortran Include Files	C/C++ Include Files
All function domains	`mkl.fi`	`mkl.h`
BLACS Routines		`mkl_blacs.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`‡
PBLAS Routines		`mkl_pblas.h`‡‡
ScaLAPACK Routines		`mkl_scalapack.h`‡‡
All Sparse Solver Routines	`mkl_solver.f90` `mkl_solver.fi`†	`mkl_solver.h`‡
PARDISO	`mkl_pardiso.f90` `mkl_pardiso.fi`†	`mkl_pardiso.h`‡
DSS Interface	`mkl_dss.f90` `mkl_dss.fi`†	`mkl_dss.h`‡
RCI Iterative Solvers ILU Factorization	`mkl_rci.fi`†	`mkl_rci.h`‡
Optimization Solver Routines	`mkl_rci.fi`†	`mkl_rci.h`‡

Vector Mathematical Functions	`mkl_vml.90`
	`mkl_vml.fi`†
	`mkl_vml.f77`
		`mkl_vml.h`‡

Vector Statistical Functions	`mkl_vsl.f90`
	`mkl_vsl.fi`†
	`mkl_vsl.f77`
		`mkl_vsl.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		

Function domain	Fortran Include Files	C/C++ Include Files
Trigonometric Transforms	`mkl_trig_transforms.f90`	`mkl_trig_transform.h`‡
Poisson Solvers	`mkl_poisson.f90`	`mkl_poisson.h`‡
Data Fitting functions	`mkl_df.f90` `mkl_df.f77`	`mkl_df.h`‡
Support functions	`mkl_service.f90` `mkl_service.fi`†	`mkl_service.h`‡
Declarations for replacing memory allocation functions. See Redefining Memory Functions for details.		`i_malloc.h`

† You can use the `mkl.fi` include file in your code instead.

‡ You can include the `mkl.h` header file in your code instead.

‡‡ Also include the `mkl.h` header file in your code.

See Also

Language Interfaces Support, by Function Domain

Support for Third-Party Interfaces

B

FFTW Interface Support

Intel® Math Kernel Library (Intel® 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® Math Kernel Library*" appendix in the Intel MKL Reference Manual for details on the use of the wrappers.

Important For ease of use, FFTW3 interface is also integrated in Intel MKL.

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
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Detailed Structure of the IA-32 Architecture Directories

Static Libraries in the `lib/ia32` Directory

File	Contents
Interface layer	
`libmkl_intel.a`	Interface library for the Intel compilers
`libmkl_blas95.a`	Fortran 95 interface library for BLAS for the Intel® 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
`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

File	Contents
`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 `lib/ia32` Directory

File	Contents
`libmkl_rt.so`	Single Dynamic Library
Interface layer	
`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
`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® Pentium®, Pentium® Pro, Pentium® II, and Pentium® III processors)
`libmkl_p4.so`	Pentium® 4 processor kernel library
`libmkl_p4p.so`	Kernel library for the Intel® Pentium® 4 processor with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), including Intel® Core™ Duo and Intel® Core™ Solo processors.
`libmkl_p4m.so`	Kernel library for processors based on the Intel® Core™ microarchitecture (except Intel® Core™ Duo and Intel® Core™ Solo processors, for which `mkl_p4p.so` is intended)
`libmkl_p4m3.so`	Kernel library for the Intel® Core™ i7 processors

File	Contents
`libmkl_vml_def.so`	Vector Math Library (VML)/Vector Statistical Library (VSL)/Data Fitting (DF) part of default kernel for old Intel® Pentium® processors
`libmkl_vml_ia.so`	VML/VSL/DF default kernel for newer Intel® architecture processors
`libmkl_vml_p4.so`	VML/VSL/DF part of Pentium® 4 processor kernel
`libmkl_vml_p4m.so`	VML/VSL/DF for processors based on the Intel® Core™ microarchitecture
`libmkl_vml_p4m2.so`	VML/VSL/DF for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
`libmkl_vml_p4m3.so`	VML/VSL/DF for the Intel® Core™ i7 processors
`libmkl_vml_p4p.so`	VML/VSL/DF for Pentium® 4 processor with Intel SSE3
`libmkl_vml_avx.so`	VML/VSL/DF optimized for the Intel® Advanced Vector Extensions (Intel® AVX)
`libmkl_vml_cmpt.so`	VML/VSL/DF library for conditional numerical reproducibility
`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® Math Kernel Library (Intel® MKL) messages in English
`locale/ja_JP/mkl_msg.cat`	Catalog of Intel MKL messages in Japanese. Available only if the Intel® C++ Composer XE or Intel® Fortran Composer XE that includes Intel MKL provides Japanese localization. Please see the Release Notes for this information.

Detailed Structure of the Intel® 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® Fortran compiler. Supports the LP64 interface

File	Contents
`libmkl_blas95_ilp64.a`	Fortran 95 interface library for BLAS for the Intel® Fortran compiler. Supports the ILP64 interface
`libmkl_lapack95_lp64.a`	Fortran 95 interface library for LAPACK for the Intel® Fortran compiler. Supports the LP64 interface
`libmkl_lapack95_ilp64.a`	Fortran 95 interface library for LAPACK for the Intel® 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® 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
`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`

File	Contents
`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 `lib/intel64` Directory

File	Contents
`libmkl_rt.so`	Single Dynamic Library
Interface layer	
`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
`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® Core™ microarchitecture
`libmkl_mc3.so`	Kernel library for the Intel® Core™ i7 processors
`libmkl_avx.so`	Kernel optimized for the Intel® Advanced Vector Extensions (Intel® AVX).
`libmkl_vml_def.so`	Vector Math Library (VML)/Vector Statistical Library (VSL)/Data Fitting (DF) part of default kernels
`libmkl_vml_p4n.so`	VML/VSL/DF for the Intel® Xeon® processor using the Intel® 64 architecture
`libmkl_vml_mc.so`	VML/VSL/DF for processors based on the Intel® Core™ microarchitecture

File	Contents
`libmkl_vml_mc2.so`	VML/VSL/DF for 45nm Hi-k Intel® Core™2 and Intel Xeon® processor families
`libmkl_vml_mc3.so`	VML/VSL/DF for the Intel® Core™ i7 processors
`libmkl_vml_avx.so`	VML/VSL/DF optimized for the Intel® Advanced Vector Extensions (Intel® AVX)
`libmkl_vml_cmpt.so`	VML/VSL/DF library for conditional numerical reproducibility
`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_blacs_intelmpi_lp64.so`	LP64 version of BLACS routines supporting Intel MPI and MPICH2
`libmkl_blacs_intelmpi_ilp64.so`	ILP64 version of BLACS routines supporting Intel MPI and MPICH2
`locale/en_US/mkl_msg.cat`	Catalog of Intel® Math Kernel Library (Intel® MKL) messages in English
`locale/ja_JP/mkl_msg.cat`	Catalog of Intel MKL messages in Japanese. Available only if the Intel® C++ Composer XE or Intel® Fortran Composer XE that includes Intel MKL provides Japanese localization. Please see the Release Notes for this information.

Detailed Directory Structure of the `lib/mic` Directory

File	Contents
Static Libraries	
`libmkl_intel_lp64.a`	LP64 interface library for the Intel compilers
`libmkl_intel_ilp64.a`	ILP64 interface library for the Intel compilers
`libmkl_blas95_lp64.a`	Fortran 95 interface library for BLAS and Intel® Fortran compiler. Supports the LP64 interface.
`libmkl_blas95_ilp64.a`	Fortran 95 interface library for BLAS and Intel Fortran compiler. Supports the ILP64 interface.
`libmkl_lapack95_lp64.a`	Fortran 95 interface library for LAPACK and Intel Fortran compiler. Supports the LP64 interface.
`libmkl_lapack95_ilp64.a`	Fortran 95 interface library for LAPACK and Intel Fortran compiler. Supports the ILP64 interface.
`libmkl_intel_thread.a`	Threading library for the Intel compilers
`libmkl_sequential.a`	Sequential library
`libmkl_core.a`	Core computation library

File	Contents
`libmkl_scalapack_lp64.a`	Static library with LP64 versions of ScaLAPACK routines
`libmkl_scalapack_ilp64.a`	Static library with ILP64 versions of ScaLAPACK routines
`libmkl_cdft_core.a`	Static library with cluster FFT functions
`libmkl_blacs_intelmpi_lp64.a`	Static library with LP64 versions of BLACS routines for Intel MPI
`libmkl_blacs_intelmpi_ilp64.a`	Static library with ILP64 versions of BLACS routines for Intel MPI
Dynamic Libraries	
`libmkl_ao_worker.so`	The Intel® MIC Architecture library to implement the Automatic Offload mode
`libmkl_intel_lp64.so`	LP64 interface library for the Intel compilers
`libmkl_intel_ilp64.so`	ILP64 interface library for the Intel compilers
`libmkl_intel_thread.so`	Threading library for the Intel compilers
`libmkl_sequential.so`	Sequential library
`libmkl_core.so`	Core computation library
`libmkl_scalapack_lp64.so`	Dynamic library with LP64 versions of ScaLAPACK routines
`libmkl_scalapack_ilp64.so`	Dynamic library with ILP64 versions of ScaLAPACK routines
`libmkl_cdft_core.so`	Dynamic library with cluster FFT functions
`libmkl_blacs_intelmpi_lp64.so`	Dynamic library with LP64 versions of BLACS routines for Intel MPI
`libmkl_blacs_intelmpi_ilp64.so`	Dynamic library with ILP64 versions of BLACS routines for Intel MPI
`locale/en_US/mkl_msg.cat`	Catalog of Intel® Math Kernel Library (Intel® MKL) messages in English
`locale/ja_JP/mkl_msg.cat`	Catalog of Intel MKL messages in Japanese. Available only if the Intel® C++ Composer XE or Intel® Fortran Composer XE that includes Intel MKL provides Japanese localization. Please see the Release Notes for this information.

Index