



$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x, y), q = \text{const} \geq 0$$

2D Helmholtz Equation

$$z_{k_1, k_2, \dots, k_d} = \sigma \times \sum_{j_d=0}^{n_d-1} \dots \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} W_{j_1, j_2, \dots, j_d} \exp\left(i 2\pi \sum_{l=1}^d j_l k_l / n_l\right)$$

General form of the Discrete Fourier Transform

Using MKL :: Intel Math Kernel Library

Intel MKL (Math Kernel Library) provides mathematical routines for scientific and engineering applications which Intel have optimised and customised for performance. The library provides sequential, threaded and distributed versions of routines. The library targets three architectures IA-32 (32bit machines), Intel 64 (AMD64/EM64T machines) and IA64 (Itanium processor family).

Introduction

Intel MKL is available on Stokes. To use the last installed version just load the intel-mkl module. This will configure your environment to use MKL for Intel 64 architecture as appropriate for Stokes. At the time of writing the default version is 10.2.1.017.

```
$ module load intel-mkl
```

MKL provides routines in the following areas:

- BLAS
- Sparse BLAS
- LAPACK
- PBLAS
- ScaLAPACK
- Sparse Solver routines
- Vector Mathematical Library functions
- Vector Statistical Library functions
- Fourier Transform functions (FFT)
- Cluster FFT
- Trigonometric Transform routines
- Poisson, Laplace, and Helmholtz Solver routines
- Optimization (Trust-Region) Solver routines
- GMP arithmetic functions

MKL supports for C/C++ and Fortran, however one should note that not all the functions are available directly for both C/C++ and Fortran. In these cases mixed language

Using MKL	
Introduction	1
Linking	1
Interface layer	2
Threading layer	2
Computational layer	2
Run-time library layer	3
Linking model	4
Static & Dynamic	5
Examples	6

techniques should be used. Intel MKL offers Fortran 90/95 and up interfaces for some routines, BLAS and LAPACK.

Linking

Intel MKL employs a layered model for linking to more easily account for differences in, threading, computation, run-time libraries (RTL) on different operating systems. The layers are:

- Interface Layer
- Threading Layer
- Computational Layer
- Run-Time Library Layer

• Interface Layer

This layer provides matching between the compiled code of an application and the threading/computational components of the library. This layer:

- provides an ILP64 interface.
- access to MKL ILP64 (access to big arrays with more than $2^{31}-1$ elements programming model).
- deals with the way in which different compilers return function values.
- a way to map single precision names to double precision names in applications that employ ILP64 programming model (e.g. Cray-style naming).

Interface Layer Libraries				
lmkl_intel_ilp64	lmkl_intel_lp64	lmkl_gf_lp64	lmkl_gf_ilp64	lmkl_intel_sp2dp

• Threading Layer

This layer helps the threaded MKL to co-operate with compiler level threading. This also provides the sequential version layer.

Threading Layer Libraries			
lmkl_intel_thread	lmkl_gnu_thread	lmkl_pgi_thread	lmkl_sequential

• Computational Layer

This is the heart of MKL and has only one variant for any processor/operating system family. The computational layer accommodates multiple architectures through identification of the architecture or architectural feature and chooses the appropriate binary code at execution. Intel MKL may be thought of as the large computational layer that is unaffected by different computational environments. Then, as it has no RTL requirements, RTLS refer not to the computational layer but to one of the layers above it: the Interface layer or Threading layer. The most likely case is matching the threading layer with the RTL layer.

Computational Layer Libraries		
lmkl_lapack	lmkl_core	lmkl_scalapack_lp64/ilp64

- **Run-Time Library Layer**

This layer has run-time library support functions. For example, libiomp and libguide are run-time libraries providing threading support for the OpenMP threading in Intel MKL. Note that when using the legacy libguide you should also link against the POSIX threads library by appending -lpthread.

In addition to the libraries provided through the layered model you have the solver libraries, Fortran 90/95 interfaces and cluster components. Each of them fits in the computational or RTL layer.

Run-Time Layer Libraries			
Solver Libraries			
mkl_solver_ilp64_sequential	mkl_solver_lp64_sequential	lmkl_solver_lp64	lmkl_solver_ilp64
Fortran 90/95 Interfaces			
lmkl_lapack95		lmkl_blas95	
Cluster Components			
lmkl_blacs_intelmpi_ilp64		libmkl_blacs_intelmpi_lp64	
lmkl_blacs_openmpi_ilp64		libmkl_blacs_openmpi_lp64	
lmkl_blacs_sgimpt_ilp64		lmkl_blacs_sgimpt_lp64	
lmkl_cdft_core		lmkl_scalapack_ilp64	
lmkl_scalapack_lp64			
FFT Interfaces			
lfftw2x_cdft_DOUBLE/SINGLE	lfftw2xc_intel/_sp		lfftw2xf_intel/_sp
	lfftw3xc_intel/_sp		lfftw3xf_intel/_sp



Sequential Version

Starting with release 9.1, the Intel MKL package provides a sequential (non-threaded) version of the library. It requires no RTL layer, that is, no Compatibility OpenMP or Legacy OpenMP run-time library, and it does not respond to the environment variable OMP_NUM_THREADS. While this version of MKL runs unthreaded code, it is thread-safe (except for LAPACK deprecated routines ?lacon, ?lasq3, and ?lasq4.), which means that you can use it in a parallel region from your own OpenMP code. You should use sequential version only if you have a particular reason not to use Intel MKL threading. The sequential version may be helpful when using MKL with programs threaded with non-Intel compilers or in other situations where you may, for various reasons, need a non-threaded version of the library. To obtain sequential version of Intel MKL, in the Threading layer, choose the _sequential library when linking. Note that the sequential library depends on the POSIX threads library (pthread), which is used to make the Intel MKL software thread-safe and should be included in the link line.

MKL Linking Model	
	[cluster components]
	[mkl_blas95] [mkl_lapack95] [FFTW Interfaces]
	[[mkl_solver, mkl_solver_lp64, mkl_solver_ilp64]]
	{mkl_intel_ilp64, mkl_intel_lp64, mkl_intel_sp2dp, mkl_gf_ilp64, mkl_gf_lp64}
	{mkl_intel_thread, mkl_gnu_thread, mkl_pgi_thread, mkl_sequential}
	[[mkl_scalapack_lp64, mkl_scalapack_ilp64]]
	[mkl_lapack] [mkl_core]
	[guide] [iomp5]
	[[blacs_(mpiLib)_lp64, blacs_(mpiLib)_ilp64]]
	[-lpthread] [-lm]
Notes: {a,b,c} only one of the libraries should be chosen, [a] library is optional, (mpiLib) stands for intelmpi, openmpi or sgimpt	



ILP64

When using ILP64 libraries the code has to be compiled with -i8 option for Fortran and -DMKL_ILP64 for C/C++. Mixing code compiled for ILP64 with LP64 libraries leads to unpredictable consequences and may lead to wrong results so pay special attention when linking or coding for ILP64. There are two categories of problems that do not support ILP64 in MKL; FFTW and GMP.

To compile against ILP64 versions (huge arrays) just replace lp64 with ilp64

Static/Dynamic Linking

Intel MKL supports both linking models static or dynamic. Each of them has their pros and cons. Static linking resolves all symbolic references at link time. The behaviour of statically built executables is predictable, because there are no run-time dependencies. The main disadvantage is that having to relink new versions of the library to your application may be error-prone and time-consuming, because you have to relink the entire application. Moreover, static linking results in large executables and uses memory less efficiently. If several executables are linked with the same library, each of them must load it into memory independently. This matters most for executables having data sizes that are small and comparable with the size of the executable.

Dynamic linking postpones the resolution of some undefined symbolic references until run time. Dynamically built executables contain those symbols along with a list of libraries that provide definitions of the symbols. When the executable is loaded, the final linking is done before the application runs. If several dynamically built executables reference the same library, it is loaded into memory only once and the executables share it, thereby saving memory. Dynamic linking enables you to separately update the libraries on which applications depend and does not require relinking the applications. The development advantages of dynamic linking are achieved at some cost to performance, because every unresolved symbol has to be looked up in a dedicated table and resolved at run time.

Linking Syntax

Dynamic Case

```
<files to link> -L$MKLPATH -I$MKLINCLUDE
[-lml_lapack95] [-lml_blas95]
[cluster components]
[{-lml_{intel, intel_ilp64, intel_lp64, intel_sp2dp, gf, gf_ilp64, gf_lp64}}]
[-lml_{intel_thread, gnu_thread, pgi_thread, sequential}]
[{-lml_solver, -lml_solver_lp64, -lml_solver_ilp64}]
[-lml_lapack] [{-lscalapack, -lscalapack_lp64, -lscalapack_ilp64}]{-lml_{ia32, em64t, ipf}}, -lml_core}
[{-liomp5, -lguide} [blacs*]] [-lpthread] [-lm]
```

Static Case

```
<files to link> -I$MKLINCLUDE
-Wl,-start-group
[$MKLPATH/libmkl_lapack95.a] [$MKLPATH/libmkl_blas95.a]
[cluster components]
[{$MKLPATH/libmkl_{intel, intel_ilp64, intel_lp64, intel_sp2dp, gf, gf_ilp64, gf_lp64}.a}
[$MKLPATH/libmkl_{intel_thread, gnu_thread, pgi_thread, sequential}.a}
[{$MKLPATH/libmkl_solver.a, $MKLPATH/libmkl_solver_lp64.a, $MKLPATH/libmkl_solver_ilp64.a}]
[{$MKLPATH/libmkl_scalapack.a, $MKLPATH/libmkl_scalapack_lp64.a, $MKLPATH/libmkl_scalapack_lp64.a }]
[$MKLPATH/libmkl_lapack.a, $MKLPATH/libmkl_{ia32, em64t, ipf}.a, $MKLPATH/libmkl_core.a}
-Wl,-end-group
[{$MKLPATH/libiomp5.a, $MKLPATH/libguide.a,}[blacs*]] [-lpthread] [-lm]
```

Notes: {a,b,c} only one of the libraries should be chosen, [a] optional, * the version matching the mpi library should be used. On stokes and stoney \$MKLPATH=\$MKLROOT/lib/em64t, \$MKLINCLUDE=\$MKLROOT/include

Linking examples

One should note the dummy libraries `mkl_intel`, `mkl_solver`, `mkl_gf`, `mkl_scalapack`, `mkl_lapack` (the last one only for the static case) are just pointing to the `_lp64` libraries. `libmkl_ia32`, `em64t` and `ipf` are dummy libraries linking to `intel_lp64`, `intel_thread` and `mkl_core`. Further details about the libraries contained by MKL can be found in the Table 3.6 from Intel MKL User's Guide. On stokes and/or stoney you will need to load the following modules `intel-cc` and/or `intel-fc`, `intel-mkl` and `mvapich2-intel` for the distributed version. The `$MKLROOT` variable is defined when the MKL module is loaded and will correspond to a given version of the MKL e.g. it may be set to: `/ichec/packages/intel/mkl/10.2.1.017`

Sequential and Threaded:

`intel_lp64`, `intel_thread`, `iomp5`, this example will link all the domain specific functions available in MKL which do not need any extra libraries or interfaces, e.g. BLAS, FFT, VSL, VML, DSS/PARDISO.

Dynamic

```
-L$MKLROOT/lib/em64t -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread
```

Static

```
$MKLROOT/lib/em64t/libmkl_intel_lp64.a -Wl,--start-group
```

```
$MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a -Wl,--end-group
```

```
-L$MKLROOT/lib/em64t -liomp5 -lpthread
```

Lapack `intel_lp64`, `intel_threads`, `iomp5`

Dynamic

```
-L$MKLROOT/lib/em64t -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_lapack -lmkl_core -liomp5 -lpthread
```

Static

```
$MKLROOT/lib/em64t/libmkl_intel_lp64.a -Wl,--start-group
```

```
$MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a -Wl,--end-group
```

```
-L$MKLROOT/lib/em64t -liomp5 -lpthread
```

Lapack `intel_lp64`, `intel_threads`, `iomp5`, Fortran 90/95 interface

Dynamic

```
-L$MKLROOT/lib/em64t/ -lmkl_lapack95 -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_lapack -lmkl_core -liomp5 -lpthread
```

Static

```
$MKLROOT/lib/em64t/libmkl_lapack95.a $MKLROOT/lib/em64t/libmkl_intel_lp64.a -Wl,
```

```
--start-group $MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a
```

```
-Wl,--end-group -L$MKLROOT/lib/em64t -liomp5 -lpthread
```

Iterative Sparse Solver intel_lp64, intel_threads, iomp5

Dynamic

```
$MKLROOT/lib/em64t/libmkl_solver_lp64.a -L$MKLROOT/lib/em64t -lmkl_intel_lp64  
-lmkl_intel_thread -lmkl_lapack -lmkl_core -liomp5 -lpthread -lm
```

Static

```
$MKLROOT/lib/em64t/libmkl_solver_lp64.a $MKLROOT/lib/em64t/libmkl_intel_lp64.a -Wl,  
--start-group $MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a  
-Wl,--end-group -L$MKLROOT/lib/em64t -liomp5 -lpthread -lm
```

cblas intel_lp64, intel_threads, iomp5

Dynamic

```
-L$MKLROOT/lib/em64t -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm
```

Static

```
$MKLROOT/lib/em64t/libmkl_intel_lp64.a -Wl,--start-group  
$MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a -Wl,--end-group  
-L$MKLROOT/lib/em64t -liomp5 -lpthread -lm
```



To use single precision fftw interfaces add the suffix `_sp` to the fftw interface name. Do not forget to add the preprocessor option `-DFFTW_ENABLE_FLOAT` at compile time and include the headers from `$MKLROOT/include/fftw`

fftw2xf interface, intel_lp64, intel_threads, iomp5

```
$MKLROOT/lib/em64t/libfftw2xf_intel.a $MKLROOT/lib/em64t/libmkl_intel_lp64.a  
-Wl,--start-group  
$MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a  
-Wl,--end-group -L$MKLROOT/lib/em64t -liomp5 -lpthread -lm
```

fftw2xc interface, intel_lp64, intel_threads, iomp5

```
$MKLROOT/lib/em64t/libfftw2xc_intel.a $MKLROOT/lib/em64t/libmkl_intel_lp64.a  
-Wl,--start-group  
$MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a  
-Wl,--end-group -L$MKLROOT/lib/em64t -liomp5 -lpthread -lm
```

fftw3xc interface, intel_lp64, intel_threads, iomp

```
$MKLROOT/lib/em64t/libfftw3xc_intel.a $MKLROOT/lib/em64t/libmkl_intel_lp64.a  
-Wl,--start-group  
$MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a  
-Wl,--end-group -L$MKLROOT/lib/em64t -liomp5 -lpthread -lm
```

fftw3xf interface, intel_lp64, intel_threads, iomp

```
$MKLROOT/lib/em64t/libfftw3xf_intel.a $MKLROOT/lib/em64t/libmkl_intel_lp64.a  
-Wl,--start-group  
$MKLROOT/lib/em64t/libmkl_intel_thread.a $MKLROOT/lib/em64t/libmkl_core.a  
-Wl,--end-group -L$MKLROOT/lib/em64t -liomp5 -lpthread -lm
```

GMP

Dynamic

```
$MKLROOT/lib/em64t/libmkl_solver_lp64.a -L$MKLROOT/lib/em64t -lmkl_intel_lp64  
-lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm
```

Static

```
$MKLROOT/lib/em64t/libmkl_solver_lp64.a $MKLROOT/lib/em64t/libmkl_intel_lp64.a  
-Wl,--start-group $MKLROOT/lib/em64t/libmkl_intel_thread.a  
$MKLROOT/lib/em64t/libmkl_core.a -Wl,--end-group -L$MKLROOT/lib/em64t -liomp5  
-lpthread -lm
```

Distributed Linking:

cdft, intel_lp64, intel_threads, iomp5, blacs

```
-Wl,--start-group $MKLROOT/lib/em64t/libmkl_cdft_core.a  
$MKLROOT/lib/em64t/libmkl_intel_lp64.a $MKLROOT/lib/em64t/libmkl_intel_thread.a  
$MKLROOT/lib/em64t/libmkl_core.a -Wl,--end-group -L$MKLROOT/lib/em64t -liomp5  
$MKLROOT/lib/em64t/libmkl_blacs_intelmpi_lp64.a -lpthread -lm
```

fdft, intel_lp64, intel_threads, iomp5, blacs

```
-Wl,--start-group $MKLROOT/em64t/libmkl_cdft_core.a  
$MKLROOT/lib/em64t/libmkl_intel_lp64.a $MKLROOT/lib/em64t/libmkl_intel_thread.a  
$MKLROOT/lib/em64t/libmkl_core.a -Wl,--end-group -L$MKLROOT/lib/em64t -liomp5  
$MKLROOT/lib/em64t/libmkl_blacs_intelmpi_lp64.a -lpthread -lm
```

scalapack

Dynamic

```
-L$MKLROOT/lib/em64t -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_lapack  
-lmkl_scalapack_lp64 -lmkl_core -liomp5 -lmkl_blacs_intelmpi_lp64 -lpthread
```

Static

```
-Wl,--start-group  
$MKLROOT/lib/em64t/libmkl_intel_lp64.a $MKLROOT/lib/em64t/libmkl_intel_thread.a  
$MKLROOT/lib/em64t/libmkl_scalapack_lp64.a $MKLROOT/lib/em64t/libmkl_core.a  
-Wl,--end-group -L$MKLROOT/lib/em64t -liomp5  
$MKLROOT/lib/em64t/libmkl_blacs_intelmpi_lp64.a -lpthread
```